Java Driving Test

Thursday 17th March 2016

9.30am – 12.30pm

THREE HOURS

(including 15 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.

There are **TWO** sections: Section A and Section B, each worth 50 marks.

Credit will be awarded throughout to successfully compiling code that is clear, concise, usefully commenting, and has pre-conditions expressed with appropriate assertions.

Important note:

- In each section, the tasks are in increasing order of difficulty. Manage your time so that you attempt both sections. You may wish to solve the easier tasks in both sections before moving on to the harder tasks
- It is critical your solution compiles for automated testing to be effective. **TEN MARKS** will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave.
- When you start Eclipse from the menu or terminal it can take 10 seconds for the splash screen to appear. You must be patient and wait for Eclipse to load. Having two copies of Eclipse running means Eclipse loses the ability to edit `java` projects. **Do not ask an invigilator for help on how to use an IDE.**
- Before you log out at the end of the test, you **must** ensure that your source code is in the correct directory otherwise your marks can suffer heavy penalties.

Section A

Problem Description

A *concurrent program* consists of a *store* (mapping variables to values) and a collection of *threads*. Each thread has a list of statements to execute. Execution of a concurrent program involves repeatedly selecting a thread whose next statement is *enabled*, and having that thread execute its next statement. This involves reading from and updating the store.

The order in which threads execute is called a *schedule*, and this order is determined by a *scheduler*.

A thread has *terminated* when it has executed all of its statements. The concurrent program has *terminated* once all threads have terminated.

Example: consider the following concurrent program:

Store:	{ x == 0, y == 0 }
Thread 0:	x = 1; y = 2;
Thread 1:	x = 2; y = 3;
Thread 2:	wait(x, 2); y = 4;

The Store contains the variables `x` and `y`. Both are initially set to zero.

Thread 0 has two *assignment* statements: `x = 1` and `y = 2`. Thread 1 has similar statements. Assignment statements are always *enabled*.

Thread 2 has a *wait* statement, `wait(x, 2)`, and an assignment statement, `y = 4`. A `wait` statement is *disabled* until its first and second arguments evaluate to the same value. Thus `wait(x, 2)` is disabled unless the value of `x` is `2`. Once enabled, execution of a wait statement does not change the contents of the store.

Execution 1: An example execution of the program is as follows:

- Thread 0 executes `x = 1`. Store becomes { `x == 1, y == 0` }.
- Thread 1 executes `x = 2`. Store becomes { `x == 2, y == 0` }.
- Thread 2 executes `wait(x, 2)`. This is possible because `x == 2` in the store, so the statement is enabled. Store remains { `x == 2, y == 0` }.
- Thread 0 executes `y = 2`. Store becomes { `x == 2, y == 2` }. Thread 0 has now terminated.
- Thread 2 executes `y = 4`. Store becomes { `x == 2, y == 4` }. Thread 2 has now terminated.
- Thread 1 executes `y = 3`. Store becomes { `x == 2, y == 3` }. Thread 1 has now terminated.
- At this point, all threads have terminated, so the program has terminated. The final result of the program is the store { `x == 2, y == 3` }.

Execution 2: An alternative execution of the same program is:

- Thread 1 executes `x = 2`. Store becomes { `x == 2, y == 0` }.
- Thread 0 executes `x = 1`. Store becomes { `x == 1, y == 0` }.
- Thread 0 executes `y = 2`. Store becomes { `x == 1, y == 2` }. Thread 0 has now terminated.
- Thread 1 executes `y = 3`. Store becomes { `x == 1, y == 3` }. Thread 1 has now terminated.
- At this point, no thread can execute: Threads 1 and 2 have terminated, and the statement `wait(x, 2)` is *disabled* for Thread 2. Since not all threads have terminated, but no threads are enabled, we say that the program has **deadlocked**.

You are given an incomplete set of classes that model programs in this simple concurrent programming language. Your task is to complete these classes, and to implement three different schedulers for execution of programs written in the language.

Getting Started

The files are located in the `concurrency` package, which has three sub-packages, `concurrency.expressions`, `concurrency.statements` and `concurrency.schedulers`.Therefore, in the `SectionA` directory in your Lexis home directory you will find the following directories:

- `/exam/SectionA/src/concurrency`
- `/exam/SectionA/src/concurrency/expressions`
- `/exam/SectionA/src/concurrency/schedulers`
- `/exam/SectionA/src/concurrency/statements`

During the test, you will be working in the following file:

- `Executor.java`: when complete, this class will control execution of a concurrent program.

You will also need to make use of (though should **NOT** modify) the following:

- `Store.java:` represents a variable store
- `expressions/*.java:` an `Expr` interface with various implementing classes
- `statements/*.java:` a `Stmt` interface with an `AssignStmt` implementation
- `Thread.java:` represents a thread
- `ConcurrentProgram.java:` represents a concurrent program
- `DeadlockException.java:` an exception to be thrown if deadlock occurs
- `schedulers/Scheduler.java:` an interface for scheduling

You should not change the signatures of any of the provided methods: auto-testing of your solution depends on the methods having exactly their original signatures. However you may feel free to add additional methods and classes (e.g. for testing) as you see fit. Any new Java files should be placed in the `SectionA` directory.

Testing

To help you test your work as you go, we have provided a test suite for most of the methods in Section A in `Progress.java`. This will not be marked.

`Progress.java` class contains some simple tests to help you gauge your progress during Section A. Most of the content of this file is commented out initially. **As you progress through the exercise you should un-comment the test method associated with each question in order to test your work.**

Any mismatches in expected and actual outputs will be reported.

The provided classes

The `Store` class, the `Expr` and `Stmt` interfaces and the implementing classes `LiteralExpr`, `IdentifierExpr` and `AssignStmt`, represent programming language variable *stores*, *statements* and *expressions*. The `Stmt` interface provides an `isEnabled(...)` method indicating whether a statement is enabled for execution. Execution of an `AssignStmt` is always enabled.

A `Store` is created, given a set of variable names. Each variable is initialized to zero.

A `Thread` has a list of statements to be executed, and an `int` *program counter* recording which statement should be executed next. `Thread` provides the following public methods:

public boolean isTerminated();
  Indicates whether the thread has any statements left to execute.

public boolean isEnabled(Store store);
  Returns `true` if and only if the next statement this thread is due to execute is *enabled* in the context of the given store.

public void step(Store store);
  Executes the next statement for the thread, and advances the thread's program counter by one. Statement execution may involve reading and updating variables in the given store. If `step` is called on a thread that has already terminated, an `UnsupportedOperationException` is thrown.

public Collection<? extends Stmt>remainingStatements();
  Returns the statements that remain to be executed by the thread. The result is a collection of some unspecified subtype of `Stmt`.

A *program*, represented by the `ConcurrentProgram` class, consists of a list of `Threads` and a `Store`. If a `Thread` is at index i in the list of `Threads` then i is the *id* of the `Thread`. `ConcurrentProgram` provides the following public methods:

public void step(int threadId);
  Causes the thread with id `threadId` to take a step.

public Set<Integer>getEnabledThreadIds();
Returns the ids of all threads that are currently enabled.

public boolean isTerminated();
Returns `true` if and only if every thread has terminated.

public int getNumThreads();
Returns the number of threads executing the program (including threads that have terminated). If `n == program.getNumThreads()` then the program has n `Threads` with ids 0, 1, . . . , n-1. This number does not change during the lifetime of a program.

public Collection<? extends Stmt> remainingStatements(int threadId);
Returns the statements that remain to be executed by the thread with id `threadId`, as a collection of some unknown subtype of `Stmt`.

Example: the method `makeExampleProgram()` in `Progress.java` (initially commented out) constructs and returns a concurrent program corresponding to the example at the start of the spec. Concurrent programs are also constructed and returned by methods `makeExampleProgram2()`, `makeExampleProgram3()` and `makeExampleProgram4()`.

A concurrent program is executed according to a *scheduler*. At each step, the scheduler decides which enabled thread should execute. The `Scheduler` interface in the `schedulers` sub-package provides a single method:

```
public int chooseThread(ConcurrentProgram program) throws DeadlockException;
```

`DeadlockException` is a direct subclass of `Exception` used to identify when deadlock has occurred.

Be careful not to accidentally delete the provided class files!

What to do

1. **Adding support for wait statements.**

   Add a new class `WaitStmt` in the `statements` sub-package, implementing the `Stmt` interface.

   - A `WaitStmt` should consist of two expressions: a left-hand-side and a right-hand side.
   - A `WaitStmt` should be *disabled* unless the left-hand-side and right-hand-side both evaluate to the same integer value.
   - Execution of a `WaitStmt` should have no effect.

   Un-comment `testQuestion1()` in `Progress.java` and check that executing this class does not lead to any **MISMATCH** reports.

   [**5 marks**]

2. **Implementing a round robin scheduler.**

   This type of scheduler executes threads in circular order. Add a class to the `schedulers` sub-package, called `RoundRobinScheduler`.

- `RoundRobinScheduler` should implement the `Scheduler` interface.
- The `chooseThread` method should throw a `DeadlockException` if no threads are enabled for the program.
- Otherwise:
  - if this is the first time that `chooseThread` has been invoked, it should return the smallest id among the enabled threads.
  - if `chooseThread` has been invoked before then let t be the id of the last thread that was scheduled. In this case, `chooseThread` should return the smallest id of an enabled thread that is larger than t. If no such id exists, `chooseThread` should return the smallest id among the enabled threads.

Un-comment `testQuestion2()` and `makeExampleProgram()`, `makeExampleProgram2()` and `makeExampleProgram3()` in `Progress.java` and check that executing this class does not lead to any **MISMATCH** reports.

[10 marks]

3. **Completing the `Executor` class.**

Look at the incomplete class `Executor`, which is built from a `ConcurrentProgram` and a `Scheduler`. The `execute` method should execute the program using the scheduler, and return a `String` that holds the final state of the program, a list of `int`s indicating the order in which threads made steps (the *history* of the execution), and a message indicating whether the program terminated gracefully or whether deadlock occurred.

You are provided with a skeleton for the `execute` method. The start of this method sets up an empty history and records initially that deadlock has not occurred. The end of this method returns a `String` result in the required format. You need to implement execution of the program in the middle part of the method.

- Execution involves repeatedly executing a step of the program until either the program terminates or deadlock occurs.
- At each step, the `Scheduler` associated with the `Executor` should be used to decide which thread should take the step.
- Each time a thread takes a step, the id of the thread should be added to the history.
- If deadlock occurs, this should be recorded by updating the `deadlockOccurred` variable, after which program execution should cease.

Un-comment `testQuestion3()` in `Progress.java` and check that executing this class does not lead to any **MISMATCH** reports.

[10 marks]

4. **Implementing a "fewest waits" scheduler.** Add at least a class to the `schedulers` sub-package, called `FewestWaitsScheduler`.

- `FewestWaitsScheduler` should implement the `Scheduler` interface.
- The `chooseThread` method should throw a `DeadlockException` if no threads are enabled for the program.
- Otherwise, `chooseThread` should return the id of the enabled thread whose remaining statements contains the smallest quantity of wait statements. In the case of a tie, return the minimum id among the enabled threads that have the fewest wait statements.
- You should avoid code duplication between the `RoundRobinScheduler` and `FewestWaitsScheduler` classes.

Un-comment `testQuestion4()` and `makeExampleProgram4()` in `Progress.java` and check that executing this class does not lead to any **MISMATCH** reports.

[15 marks]

5. **Fixing equality testing.**

Let us say that two `Executor` objects are equal if their programs have identical string representations, according to the `toString()` method of the `ConcurrentProgram` class.

At the end of the `Executor` class you will find an erroneous attempt to override the equals method of object to provide this notion of equality between `Executor` objects.

This attempt to override equals is **wrong** for several reasons. Identify why the attempt is wrong and rectify it.

No explicit tests are provided for this question.

[10 marks]

Total for Section A: 50 marks

Section B

Problem Description

Many problems can be solved using graphs and graph algorithms. The *Weighted Nine Tails Problem* is one example. Nine coins are placed in a 3×3 matrix configuration with some face up (heads) and some face down (tails). The aim is to find the minimum number of flips that leads to all coins being face down (the **target configuration**). A legal move takes a *face-up* coin and flips it over together with the coins immediately adjacent (but not diagonal to it) regardless of whether they are face up or down. The **weight** (or cost) of a move is the number of coins that have been flipped in that move. A solution is a sequence of legal moves leading from a given configuration to the target configuration. An optimal solution is one where the total number of flips is the smallest.

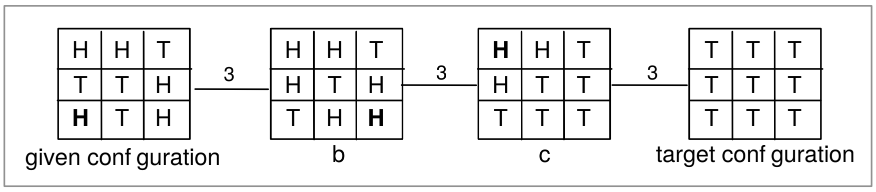

Figure 1: Configuration

For example, Figure 1 shows the optimal solution from the given configuration to the target configuration. The first move starts by flipping the coin in the bottom left which causes both the coin immediately above it and the coin immediately to its right to flip. The result is shown in `b`. Now flipping the coin in the bottom right corner of `b` causes both the coin immediately above it and the coin immediately to its left to flip. The result of this is `c`. Now flipping the top-left coin and its two adjacent coins attains the goal of all coins being tails up. The three moves each results in three coins being flipped (a weight of 3), giving a total weight of 9.

The *Weighted Nine Tails Problem* is interesting because it can be reduced to solving the more general problem of finding the cheapest path (the optimal solution) between two vertices in an edge-weighted graph.

You need to complete the provided code that solves the *Weighted Nine Tails Problem* described above. The program:

1. constructs a **nine tails weighted graph** that stores **all** possible configurations and legal moves with their respective weights (the method `constructGraph` of the class `NineTailsWeightedGraph`);
2. generates a **minimum spanning tree** from the *target configuration* to all the configurations (vertices) in the graph. This step also computes the *weight* of each path, that is the number of flips needed from the vertex at the end of the path back to the target configuration (the `constructMinimumSpanningTree` method of class `NineTailsWeightedGraph`);
3. prompts the user for a starting configuration as a `String` of 9 '`H`' and '`T`' characters;
4. finds (using the minimum spanning tree) the **cheapest path** from the user entered starting configuration to the target configuration, then prints this path together

with the total number of flips needed (see the `printShortestPath` method of the class `NineTailsWeightedGraph`).

In a **nine tails weighted graph**, vertices are indexed from 1 to 512, as there are 512 (2^9) possible configurations of 9 coins. Given any configuration, it is possible to generate its index as an integer; and, given any index, it is possible to generate its configuration (methods `configurationToIndex` and `indexToConfiguration` respectively).

Edges are instances of an inner class `WeightedEdge`. An edge stores links between two configurations, the *parent* and *child*, and the weight of the move from parent to child. The parent is the configuration to which the legal move is applied, while the child is the configuration resulting from that move. For instance, from Figure 1, consider the edge between configuration `c` (whose index is 96) and the target configuration (whose index is 512). Since the legal move is applied to configuration `c`, the link between these two configurations is represented by an edge object where the parent is 96, the child is 512 and the weight is 3 (since three flips are made in this move).

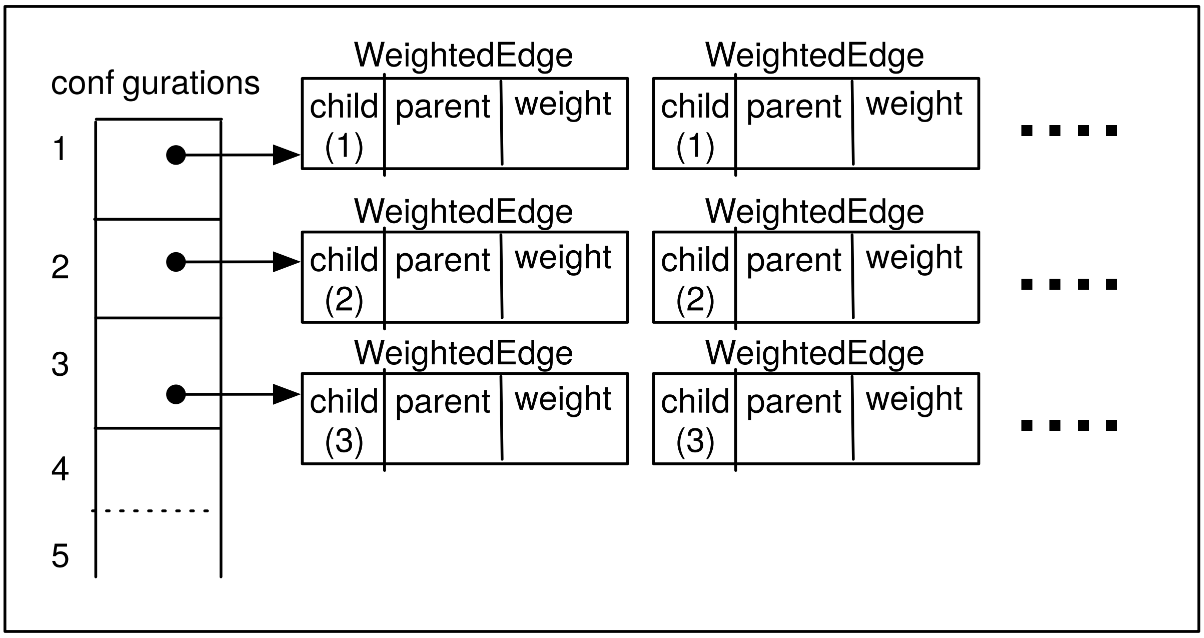

Figure 2: NineTailsWeightedGraph

A neat way to represent a nine tails weighted graph is as a list of coin configurations where each element in that list is a priority queue of weighted edges, prioritised by weight: the lower the weight, the higher the priority. The child of a weighted edge in the nth priority queue is always n. The size of the configurations list is the number of vertices in the graph. See Figure 2.

Minimum Spanning Tree Algorithm

The construction of the minimum spanning tree starts from the *target configuration* (the root) and computes the tree of minimum (cheapest) paths from this configuration to all the other configurations in the graph. Given the chosen implementation of the nine tails weighted graph, the minimum spanning tree can simply be stored in an array of configuration indices. The size of the array (called *nextMoves*) is equal to the number of vertices in the graph. An array index corresponds to a parent configuration index. Given an array index (parent configuration index) the value is the chosen child configuration index as the next best move.

The minimum spanning tree can be constructed using the following algorithm, where s denotes the index of the target configuration, V the set of vertices in the graph, and $w(u, v)$ denotes the weight of the edge between u and v, where u is the child configuration and v is the parent:

```
constructMinimumSpanningTree()
 Let nextMoves be the array storing the minimum spanning tree
 Let visited be the set of visited vertices in the graph
 Let costs be the array of the minimum weight from a parent configuration to $s$ computed
     so far

 Initially visited contains just $s$
 costs[$s$] = 0
 nextMoves[$s$] = -1       (meaning no next move from $s$)

 while (size of visited <size of $V$) {
      find a triple $(u, v, c)$ where $u$ and $v$ are configuration indices and $c$ is a cost such
          that:
           1. $v$ is a not-yet-visited parent of $u$
           2. $c$ is costs[$u$] + $w(u, v)$
           3. there is no other triple $(u', v', c')$, with $u'$ in the visited set, such that $c' < c$
      add $v$ to visited
      costs[$v$] = $c$
      nextMoves[$v$] = $u$
 }
```

Getting Started

The files used in Section B can be found in the `SectionB` directory in your Lexis home directory:

`/exam/SectionB/src/`

Also a UML class diagram describing the architecture of the system is provided (See Figure 3, page 12).

During Section B, you will be working in the following files:

- `ListArrayBased.java`: class that implements `ListInterface`. You will need to complete a method in the class.
- `NineTailsWeightedGraph.java`: class that uses a graph for the weighted nine tails problem. This class includes inner classes. It also has auxiliary methods for printing and for generating: legal moves, index from configuration, and configuration from index. It also includes the auxiliary method `constructGraph` for constructing a nine tail weighted graph, and `constructMinimumSpanningTree` for constructing the minimum spanning tree. You will have to implement these two methods.
- `PriorityQueue.java`: class that implements `PriorityQueueInterface`. You will need to complete two methods in this class.

You may also need to make use of (though you should **NOT** need to edit) the following:

- `WeightedNineTailsProblem.java`: this class creates a graph, reads the initial configuration of coins, generates the index of the initial configuration and prints the shortest path from this configuration to the target configuration.
- `ListIndexOutOfBoundsException.java`: this class implements the out of bounds exception for a List class.
- `PQException.java`:this class implements the PQ exception.
- `PriorityQueueInterface.java`: adds a new entry to the priority queue according to the priority value.
- `ListInterface.java`: represents a collection of abstract methods to be used by a List class.

What to do

Your task is to complete the provided implementation. You must also carefully read the commented source files.

1. `add(int givenPosition, T newItem)` for the class `ListArrayBased<T>`.

   [7 marks]

2. `add(T newEntry)` and `PQRebuild(int root)` for the class `PriorityQueue<T extends Comparable<T>>`.

   [13 marks]

3. `constructGraph()` for the class `NineTailsWeightedGraph` that constructs a nine tails weighted graph as described above.

   [12 marks]

4. `constructMinimumSpanningTree()` for the class `NineTailsWeightedGraph` that computes the minimum spanning tree as described in the Minimum Spanning Tree Algorithm section.
   [18 marks]

   Total for Section B: 50 marks

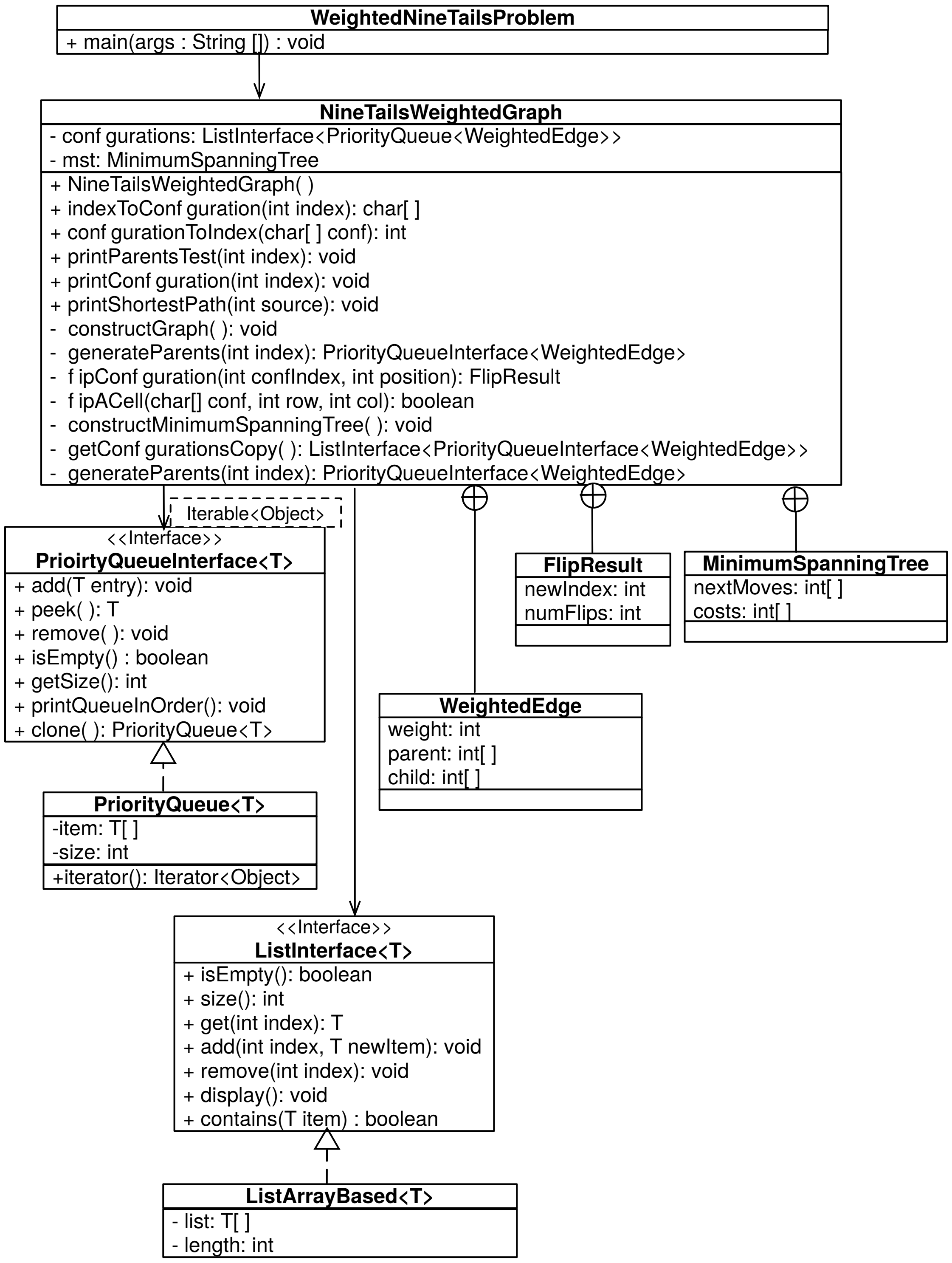
WeightedNineTailsProblem
+ main(args : String []) : void
NineTailsWeightedGraph
- conf gurations: ListInterface<PriorityQueue<WeightedEdge>>
- mst: MinimumSpanningTree
+ NineTailsWeightedGraph( )
+ indexToConf guration(int index): char[ ]
+ conf gurationToIndex(char[ ] conf): int
+ printParentsTest(int index): void
+ printConf guration(int index): void
+ printShortestPath(int source): void
- constructGraph( ): void
- generateParents(int index): PriorityQueueInterface<WeightedEdge>
- f ipConf guration(int confIndex, int position): FlipResult
- f ipACell(char[] conf, int row, int col): boolean
- constructMinimumSpanningTree( ): void
- getConf gurationsCopy( ): ListInterface<PriorityQueueInterface<WeightedEdge>>
- generateParents(int index): PriorityQueueInterface<WeightedEdge>
Iterable<Object>
<<Interface>>
PrioirtyQueueInterface<T>
+ add(T entry): void
+ peek( ): T
+ remove( ): void
+ isEmpty() : boolean
+ getSize(): int
+ printQueueInOrder(): void
+ clone( ): PriorityQueue<T>
PriorityQueue<T>
-item: T[ ]
-size: int
+iterator(): Iterator<Object>
FlipResult
newIndex: int
numFlips: int
MinimumSpanningTree
nextMoves: int[ ]
costs: int[ ]
WeightedEdge
weight: int
parent: int[ ]
child: int[ ]
<<Interface>>
ListInterface<T>
+ isEmpty(): boolean
+ size(): int
+ get(int index): T
+ add(int index, T newItem): void
+ remove(int index): void
+ display(): void
+ contains(T item) : boolean
ListArrayBased<T>
- list: T[ ]
- length: int

Figure 3: System Architecture

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning