# Claremont McKenna College
# Computer Science

## CS 62 Handout 8: PS 6 October 28, 2015

This problem set is due 11:55pm on Sunday, November 8.

**Be sure to include** a comment at the top of each file submitted that gives your name and email address.

Submit the following files to the **Assignments** area on **Sakai**. Multiple submissions are allowed within the time limit. Please do **not** submit `.class` files or Eclipse project files.

```
SLL.java
UseSLL.java
DLL.java
UseDLL.java
Point.java
TwoQS.java,
InfixEval.java,
UseInfixEval.java
BankSim.java
UseBankSim.java
(Readme.txt if needed, and any other files that make up your
 solution.  Your submission should be self-sufficient in that
 the grader will be able to run your program without having to
 add any additional file.)
```

## Problem 1 (20 points)

We studied an implementation of a singly linked list in class (see `SLL.java`). In this problem you will revise it to satisfy the following requirements:

- In the given implementation of `SLL`, elements were kept in reverse order. That is, `head` is pointing to the node containing the last element added. Your new implementation is to keep them in correct order, i.e., `head` will point to the node containing the first element added to the list. Add an additional field named `tail` to implement this new feature. With this new implementation `head` will always point to the node containing the element that was added the earliest; `tail` will always point to the node containing the element that was added most recently. Your add method should still have its running time complexity of $\Theta(1)$. You will have to revise each method that has already been implemented in `SLL` to work correctly with this new requirement.
- Make the `size` method *recursive*. You may find introducing an auxiliary method useful. Make sure you use appropriate visibility modifier if you are introducing an auxiliary method. If you introduce an auxiliary method named `sizeAux`, the method `size` would not be recursive, but `sizeAux` would be.

- Also make the `contains` method recursive. If you want to introduce any auxiliary method(s), do so.
- Implement the `indexOf(Object o)` method. In the original implementation, it is throwing an exception of type `UnsupportedOperationException` and you should replace that body with a correct implementation. If you are not sure what this method is supposed to do, check the Java documentation. Use recursion in this method as well.
- Implement `get(int index)` and `set(int index, E element)` also. They do not have to use recursion, but you are welcome to use it if you wish.
- Implement `Object[] toArray()` also using iteration or recursion. Your call.
- For the iterator, you will have to revise its implementation to be consistent with the changes if necessary.
- Create a class named `UseSLL` in a file named `UseSLL.java`. Add a `main` method there to test your new implementation in `SLL`. Make sure you include adequate tests for each method that you revised.
- For each method you revised, state the running time complexity in $\Theta$ notation as a comment just above the method's header.

For this problem hand in `SLL.java` and `UseSLL.java`.

## Problem 2 (20 points)

This time you will implement a doubly linked list. Let's call it `DLL` in a file named `DLL.java`. I suggest that you start with `SLL.java` that we studied in class and start modifying it to make it implement a doubly linked list. Your implementation is to satisfy the following:

- Include both `head` and `tail` as described in Problem 1 above.
- Iterator for `DLL` would have to implement the `java.util.ListIterator` interface rather than `java.util.Iterator`. Your implementation should include at least `hasNext`, `hasPrevious`, `next`, and `previous`. You are welcome to include an implementation of `nextIndex` and `previousIndex` if you wish.
- The methods that you wrote using recursion in Problem 1 should also be recursive in this class.
- Use `UseDLL.java` provided in the [given] link. To the given `UseDLL.java` you should add more test cases to test other functions that you implemented in your solution.
- For each method that you implemented, state the running time complexity in $\Theta$ notation as a comment just above the method's header.

For this problem hand in `DLL.java` and `UseDLL.java`.

## Problem 3 (10 points)

Implement a stack using two queues. More specifically, you are to implement a class named `TwoQS` in a file named `TwoQS.java` that implements the `Stack` interface given in [given]. The class `TwoQS` should have *only* two fields, `q1` and `q2`, which are declared as follows:

```
private Queue<???> q1;
private Queue<???> q2;
```

and can be initialized as objects of the class `QueueArray`. Modify the parameters appropriately. Since these are the only fields available in the class `TwoQS`, you will have to maintain the entries in one of the queues. Think

about why you would need another queue. Each method that you implement of course may have its own local variables though.

You will recognize that this is not an efficient way to implement a stack, but it will help you understand how both stacks and queues work by doing this exercise.

Use any of the queue and stack related classes provided in the [given] folder as you need them. Obviously you will need to implement the `Stack` interface in `Stack.java` and you will need other classes to implement it.

Add a `main` in your `TwoQS` class to demonstrate that your implementation of stack using two queues works correctly. Make your `main` brief but sufficient.

Submit `TwoQS.java` along with whatever other files that we will need to run your solution.

# Problem 4 (20 points)

We can evaluates algebraic expressions using stacks. In this problem you will implement an algorithm that converts an infix expression to a postfix expression first. Then, implement another algorithm that evaluates the converted postfix expressions to get the results.

Name your class `InfixEval` in a file named `InfixEval.java`. Also use two methods of the following signatures:

```
public static String[] toPostfix(String[] infix);
public static double evalPostfix(String[] postfix);
```

Your program should support at least the following operators: $+$, $-$, $\times$, and $/$. Also ( and ) for grouping. Optionally you may add more operators. Although `toPostfix` accepts an array of strings as an argument, your `main` should start with a string of infix expression, e.g., `"456 + 34 - 2"`, which will first be turned into an array of strings before it is passed to `toPostfix`. Add a `main` that tests your implementation with enough examples that demonstrate that your program is working correctly. Include some error cases in your tests. You may assume that operands are integers. Of course, a grouped expression can be an operand. You may also format your input string for ease of processing, e.g., you may use spaces as delimiters separating operands, operators, and the grouping symbols.

Since the class `InfixEval` would contain only static methods, let us define a default constructor with the visibility declared to be `private`. When you create a class with only static methods and fields, that technique is often used to prevent instances from being created.

Hand in `InfixEval.java` along with `UseInfixEval.java` that tests your implementation.

# Problem 5 (50 points)

You are allowed to do *pair programming* on this problem *only*. By pair programming I mean you work with one other person. Copying your partner's code is *not* pair programming. In pair programming two programmers sit in front of a single keyboard and start programming. At any point in time during the entire programming process both are paying attention to the task at hand and whoever wants to type grabs the keyboard and types. Two people are acting as if there is only one person programming.

When you submit, only one of the partners should submit the solution files for this problem. The other person should just submit an empty file named `UseBankSim.java` with only two names added as a comment in the file. Your partner must be a student in CS 62. If you choose to do it alone, that is fine too.

In this problem you will write a program that simulates a bank teller window. The kind of simulation that you will implement is called an *event-driven simulation*. The goal is to reflect the long-term average behavior of the system rather than to predict occurrences of specific events.

Although the technique for generating events to reflect the real world is interesting and important, it requires a good deal of mathematical sophistication. Therefore, simply assume that you already have a list of events available

for our use. That is, assume that you have a file containing the time of each customer's arrival–an **arrival event**–and the duration of that customer's transaction once the customer reaches the teller. For example, a file containing the data

```
20      5
22      4
23      2
30      3
```

indicates that the first customer arrives 20 minutes into the simulation and that the transaction–once begun–requires 5 minutes; the second customer arrives 22 minutes into the simulation and the transaction requires 4 minutes; and so on. Assume that the input file is ordered by arrival time. Notice that the file does not contain **departure events**. The simulation can easily compute them.

You could conduct a simulation by hand with the previous example data as follows:

```
 Time          Event
-------    ------------------------------------------------------
  20       Customer 1 enters bank and begins transaction
  22       Customer 2 enters bank and stands at end of line
  23       Customer 3 enters bank and stands at end of line
  25       Customer 1 departs; customer 2 begins transaction
  29       Customer 2 departs; customer 3 begins transaction
  30       Customer 4 enters bank and stands at end of line
  31       Customer 3 departs; customer 4 begins transaction
  34       Customer 4 departs
```

A customer's wait time is the elapsed time between arrival in the bank and the start of the transaction. *The average of this wait time over all the customers is the statistic that you want to obtain.*

This simulation is concerned with two types of events:

- **Arrival events:** These are *external events* provided by an input file. When a customer arrives at the bank, one of two things happens. If the teller is idle when the customer arrives, the customer enters the line and begins the transaction immediately. If the teller is busy, the new customer must stand at the end of the line and wait for service.
- **Departure events:** These events indicate the departure from the bank of a customer who has completed a transaction. These are *internal events* computed by the simulation. When a customer completes the transaction, he or she departs and the next person in line, if there is one, begins a transaction.

The main tasks of an algorithm that performs the simulation are to determine the times at which the events occur and to process the events when they do occur. The algorithm can be stated at a high level as follows:

```
// Initialize the line to "no customers"
while (events remain to be processed) {
    currentTime = time of next event
    if (event is an arrival event) {
        process the arrival event
    }
    else {
        process the departure event
    }
    // When an arrival event and departure event occur
    // simultaneously, arbitrarily process the arrival event first
```

To be able to determine the `time of next event`, you must maintain an *event list*. An event list contains all arrival and departure events that will occur but have not occurred yet. The times of the events in the event list are in ascending order, and thus the next event to be processed is always at the beginning of the list. The algorithm simply gets the event from the beginning of the list, advances the time specified, and processes the event. The difficulty, then, lies in successfully managing the event list.

Since each arrival event generates exactly one departure event, you might think that you should read the entire input file and create an event list of all arrival and departure events sorted by time. That approach would be impractical. Why? As you will see, you can instead manage the event list for this particular problem so that it always contains at most one event of each kind.

Recall that the arrival events are specified in the input file in ascending time order. You thus never need to worry about an arrival event until you have processed all the arrival events that precede it in the file. You simply keep the earliest unprocessed arrival event in the event list. When you eventually process this event–that is, when it is time for this customer to arrive–you replace it in the event list with the next unprocessed arrival event, which is the next item in the input file.

Similarly, you need to place only the next departure event to occur on the event list. But, how can you determine the times of the departure events? Observe that the next departure event always corresponds to the customer that the teller is currently serving. As soon as a customer begins service, the time of his or her departure is simply

```
time of next departure = time service begins + length of transaction
```

As soon as a customer begins service, you place a departure event corresponding to this customer in the event list.

Now, consider how you can process an event when it is time for the event to occur. You must perform two general types of actions:

- **Update the line:** Add or remove customers.
- **Update the event list:** Add or remove event.

As customers arrive, they go to the back of the line. The current customer, who is at the front of the line, is being served, and it is this customer that you remove from the system next. It is thus natural to use a queue to represent the line of customers in the bank. For this problem the only information that you must store in the queue about the customer is the time of arrival and the length of transaction. The event list, since it is sorted by time, is not a queue. We will examine it in more detail shortly.

To summarize, you process an event as follows:

```
To process an arrival event

    // Update the event list
    delete the arrival event for customer C from the event list
    if (new customer C begins transaction immediately) {
        insert a departure event for customer C into the
            event list (time of event = current time + transaction
            length)
    }
    if (not at the end of the input file) {
        read a new arrival event and add it to the event list
        (time of event = time specified in file)
    }
```

Because a customer is served while at the front of the queue, a new customer always enters the queue, even if the queue is empty. You then delete the arrival event for the new customer from the event list. If the new customer

is served immediately, you insert a departure event in the event list. Finally, you read a new arrival event into the event list. This arrival event can occur either before or after the departure event.

```
To process a departure event

    // Update the line
    delete the customer at the front of the queue
    if (the queue is not empty) {
        the current front customer begins transaction
    }

    // Update the even list
    delete the departure event from the event list
    if (the queue is not empty) {
        insert into the event list the departure event for the
        customer now at the front of the queue (time of
        event = current time + transaction length)
    }
```

After processing the departure event, you do not read another arrival event from the file. Assuming that the file has not been read completely, the event list will contain an event whose time is earlier than any arrival still in the input file.

Examining the event list more closely will help explain the workings of the algorithm. There is no typical form that an event list takes. For the simulation, however, the event list has four possible configurations:

1. Initially, the event list contains an arrival event $A$ after you read the first arrival event from the input file but before you process it:

   ```
   Event list: A (initial state)
   ```

2. Generally, the event list for this simulation contains exactly two events: one arrival event $A$ and one departure event $D$. Either the departure event is first or the arrival event is the first as follows

   ```
   Event list: D A (general case - next event is a departure)
   or
   Event list: A D (general case - next event is an arrival)
   ```

3. If the departure event is first and that event leaves the teller's line empty, a new departure event does not replace the just-processed event. Thus, in this case, the event list appears as:

   ```
   Event list: A (a departure leaves the teller's line empty)

   Notice that this instance of the event list is the same as its
   initial state.
   ```

4. If the arrival event is first and if, after it is processed, you are at the end of the input file, the event list contains only a departure event:

   ```
   Event list: D (the input has been exhausted)
   ```

Other situations result in an event list that has one of the previous four configurations.

You insert new events either at the beginning of the event list or at the end, depending on the relative times of the new event and the event currently in the event list. For example, suppose that the event list contains only an arrival event $A$ and that another customer is now at the front of the line and beginning a transaction. You need to generate a departure event $D$ for this customer. If the customer's departure time is before the time of the arrival event $A$, you must insert the departure event $D$ before the event $A$ in the event list. However, if the departure time is after the time of the arrival event, you must insert the departure event $D$ after the arrival event $A$. In the case of a tie, you need a rule to determine which event should take precedence. In our solution, we arbitrarily choose to place the departure event after the arrival event.

You can now combine and refine the pieces of the solution into an algorithm that performs the simulation by using the ADT queue operations to manage the bank line:

```
// Performs the simulation.
simulate() {
    Create an empty queue bankQueue to represent the bank line
    Create an empty event list eventList

    Get the first arrival event from the input file
    Place the arrival event in the event list

    while (the event list is not empty) {
        newEvent = the first event in the event list
        if (newEvent is an arrival event) {
            processArrival(newEvent, arrivalFile, eventList, bankQueue)
        }
        else {
            processDeparture(newEvent, eventList, bankQueue)
        }
    }
}

// Processes an arrival event
processArrival(Event arrivalEvent, File arrivalFile,
               EventList anEventList, Queue bankQueue) {
    atFront = bankQueue.isEmpty()  // Present queue status

    // Update the bankQueue by inserting the customer, as
    // described in arrivalEvent, into the queue
    bankQueue.enqueue(arrivalEvent)

    // Update the event list
    Delete arrivalEvent from anEventList

    if (atFront) {
        // The line was empty, so new customer is at front of line
        // and begins transaction immediately
        Insert into the anEventList a departure event that corresponds
            to the new customer and has
            currentTime = currentTime + transaction length
    }
    if (not at end of input file) {
        Get the next arrival event from arrivalFile
        Add the event - with time as specified in the input file - to
```

```
            anEventList
    }
}

// Processes a departure event
processDeparture(Event departureEvent, EventList anEventList,
                 Queue bankQueue) {
    // Update the line by deleting the front customer
    bankQueue.dequeue()

    // Update the event list
    Delete departureEvent from anEventList
    if (!bankQueue.isEmpty()) {
        // Customer at front of line begins transaction
        Insert into anEventList a departure event that corresponds to
            the customer now at the front of the line and has
            currentTime = currentTime + transaction length
    }
}
```

The following begins a trace of this algorithm for the data given earlier and shows the changes to the queue and event list. I strongly suggest you complete the trace to enhance your understanding on how it works. In the trace, I am using shorthand notations: `BQ` stands `bankQueue`, `EL` stands for `anEventList`, and `Ck` stands for the customer `k`.

| Time | Action | bankQueue (BQ) | anEventList (EL) |
|---|---|---|---|
| 0 | Read file; | (empty) | [A.20.5] |
| | Place event in EL | | |
| 20 | Update EL and BQ | [20.5] | (empty) |
| | C1 enters bank | | |
| | C1 begins transaction | [20.5] | [D.25] |
| | Create departure event | | |
| | Read file; | [20.5] | [A.22.4][D.25] |
| | Place event in EL | | |
| 22 | Update EL and BQ | [20.5][22.4] | [D.25] |
| | C2 enters bank | | |
| | Read file; | [20.5][22.4] | [A.23.2][D.25] |
| | place event in EL | | |
| 23 | Update EL and BQ | [20.5][22.4][23.2] | [D.25] |
| | C3 enters bank | | |
| | Read file; | [20.5][22.4][23.2] | [D.25][A.30.3] |
| | place event in EL | | |
| 25 | Update EL and BQ | [22.4][23.2] | [A.30.3] |
| | C1 departs | | |
| | C2 begins transaction; | [22.4][23.2] | [D.29][A.30.3] |
| | Create departure event | | |

The event list is, in fact, an ADT itself. By examining the previous pseudocode, you can see that this ADT must include at least the following operations:

```
// Creates an empty event list.
void createEventList()

// Determines whether an event list is empty.
boolean isEmpty()

// Inserts anEvent into an event list so that events are ordered by
// time.  If an arrival event and a departure event have the same
// time, the arrival event precedes the departure event.
void insert(Event anEvent)

// Deletes the first event from an event list.
void delete()

// Retrieves the first event in an event list.
Event retrieve()
```

Now, your task is to implement the event-driven simulation described so far. A queue of arrival events will represent the line of customers in the bank. Maintain the arrival events and departure events in an ADT event list, sorted by the time of event.

The input is a test file of arrival and transaction times. Each line of the file contains the arrival time and required transaction time for a customer. The arrival times in the input file are sorted by time in increasing order.

Your program must count customers and keep track of their cumulative waiting time. These statistics are sufficient to compute the average waiting time after the last event has been processed.

Display a trace of the events executed and a summary of the computed statistics (total number of arrivals and average time spent waiting in line). For example, the input file shown in the left columns of the following table should provide the output shown in the right column.

```
Input File   Output
----------   ----------------------------------------
 1    5      Simulation begins
 2    5      Processing an arrival event at time:    1
 4    5      Processing an arrival event at time:    2
20    5      Processing an arrival event at time:    4
22    5      Processing a departure event at time:   6
24    5      Processing a departure event at time:  11
26    5      Processing a departure event at time:  16
28    5      Processing an arrival event at time:   20
30    5      Processing an arrival event at time:   22
88    3      Processing an arrival event at time:   24
             Processing a departure event at time:  25
             Processing an arrival event at time:   26
             Processing an arrival event at time:   28
             Processing an arrival event at time:   30
             Processing a departure event at time:  30
             Processing a departure event at time:  35
             Processing a departure event at time:  40
             Processing a departure event at time:  45
             Processing a departure event at time:  50
```

```
Processing an arrival event at time:   88
Processing a departure event at time:  91
Simulation ends

Final Statistics:
Total number of customers processed: 10
Average amount of wait time:         5.6
```

Use `java.util.LinkedList` as a queue rather than using one of our implementations. If you read the document on `java.util.Queue`, you will realize that the `java.util.Queue` interface is implemented by many classes, one of which is `java.util.LinkedList`. If you read the documentation on `java.util.LinkedList`, you will see what to do to use it. You will notice that the function names in that interface are different from the ones we used for our queue implementation.

Use a linked list implementation for the ADT event list. You will have to implement this linked list yourself.

Note that the pseudocode used in the problem description suggests some names, but you are not required to use the same names. In fact, it may not be appropriate to use some of those names, e.g., you would not create a method named `createEventList`. Instead use a constructor for the class that implements the event list ADT. Also I hope the semantics of the pseudocode used is clear enough. If not, please ask.

Create a class named `BankSim` in a file named `BankSim.java` and use it as your main class. Create `UseBankSim.java` to test your implementation of the simulation. Of course, you will create additional classes that make up your solution. Submit all the files including your input files so that the grader will be able to run your program without having to add any additional file(s).

**Clarification:** As you translate the pseudocode given above into Java, you will have to modify the signature of a method given above. Remember that the code given above is pseudocode, not Java. For example, in the pseudocode some information may be passed into a method as a parameter, but it may make a better sense not to in Java if you find a better way of sharing that information between methods. The type names I used in the pseudocode may not be exact, for example, I used `Queue` as one of the types, but you will end up using `java.util.Queue<SomeType>` or even `java.util.LinkedList<SomeType>` instead. You get the idea.

I added the pseudocode and other data used above to the [given] folder in case you want to save some typing.