# CPS 305 Project Two: Linked Lists From Scratch

## Rationale (or, how I learned to stop worrying and love the low level array representations)

High level languages and data structures enable us to allocate our precious brain cycles to think about the actual problem that we need to solve instead of dealing with the nitty gritty details of what is really going on. Let the computer and the virtual machine take care of all that grunt work!

The normal way to represent linked lists and other data structures of individual nodes pointing to one another is to define two separate classes; one whose objects represent **the list as a whole**, and another (typically defined as an *inner class* of the previous class) whose objects represent **the individual nodes** that contain references to other nodes. This approach has several upsides, especially when combined with object-oriented programming techniques of *polymorphism* and *generics*. As for all objects in Java, *garbage collection* will then automatically release the objects that have become unreachable, so we don't need to worry about doing that in our code.

However, such a high level representation has one giant downside of its overhead of memory consumption. Consider a simple linked list whose keys are integers. Both the *dynamic binding of method calls* and the implementation of memory management and garbage collection in the Java Virtual Machine necessitate every object to have constant bookkeeping memory overhead in addition to the space actually needed to store the data in the object fields. This bookkeeping cost is separate for each individual object. This makes it far worse to store million individual node objects compared to storing one array object (with the array object bookkeeping data stored only once) that contains the same data payload as those nodes.

These days our computers and JVM processes have such an embarrassment of riches of heap memory available compared to the size of the problem that the above is not really an issue for realistic problem sizes. But suppose that your linked lists grew up to contain, say, a hundred million elements, constantly being allocated and garbage collected. It would make a pretty huge difference whether that list structure uses 800 megabytes or about 2.5 gigabytes of the process RAM.

In this second programming project, we use Java to take a time travel journey back to the sixties and forget all the high level object oriented stuff. Instead of storing the nodes of a list as separate objects, we use arrays to simulate the memory in which these node objects are stored. For example, The Art of Computer Programming still presents all its algorithms and data structures in such low-level way. Even though the course CPS 305 is no longer given in C where this sort of explicit low level memory management is absolutely necessary, you should still learn this kind of thinking to be able to do such low level optimizations in code, because sometimes they can really pay off even in our terabyte age. The purpose of this project is to give you that knowledge.

# The IntList Class

The instructors provide the class IntList.java that implements a mechanism to manage **chains** of singly linked nodes that contain `int` keys. There are no header nodes to represent the entire list, but all operations consists of management of chains. **You may not modify this class in any way in this project**, but you should still read through its code carefully to get an understanding of what is going on. The integer arrays `key` and `next` are used to store the key and the successor of each node. These two arrays essentially simulate the random access memory in which these nodes are stored with zero additional overhead. Therefore each node is identified by its integer index to these arrays, instead of an actual memory address stored in a reference variable.

To initialize the `Intlist` class and actually allocate the memory for the arrays used to store the nodes, the user code must first call the method `IntList.initialize(m)` with the maximum number of nodes as the parameter `m`. (This call is made by the automated tester at the startup, so your code does not need to worry about that. But just in case that somebody other than the instructor will ever use this class as part of their own project that needs lots of little chains...)

The relevant `public static` methods of `IntList` that your code will have to call are `getKey`, `getNext`, `setNext`, `allocate` and `release`. None of your methods should ever need to call the method `setKey`, but this method is provided here again in case that somebody wants to use `IntList` as part of some real world project. The following table shows how to convert the ordinary linked list operations on separate `Node` objects into the `IntList` method calls used in this project:

| **Operation** | **`Node` objects, usual way** | **But in this project...** |
|---|---|---|
| Declare a variable `n` used to refer to one node | `Node n;` | `int n;` |
| Create a new node | `n = new Node(k);` | `n = IntList.allocate(k);` |
| Get key in node | `n.key` | `IntList.getKey(n)` |
| Assign key `k` into node | `n.key = k;` | `IntList.setKey(n, k);` |
| Get successor of node | `n.next` | `IntList.getNext(n)` |
| Assign successor `m` to node | `n.next = m;` | `IntList.setNext(n, m);` |
| Check if node exists | `n != null` | `n != 0` |
| Release an unused node to make it available for future node allocations | *(The Java garbage collection takes care of that for you, and you don't do anything)* | `IntList.release(n);` |

Note that in this representation, only individual nodes exist (once again, these nodes don't really finger-quotes "exist" as actual Java objects, but are simulated inside two ordinary integer arrays), and there is no separate data type whose objects would represent header nodes to represent an entire list as a whole. Therefore, the individual node at the head of some chain simultaneously stands for the entire chain hanging from that node. Methods that operate on entire lists receive the integer index of this head node as parameter.

Unlike the Java object heap with its additional bookkeeping data per object, this representation of singly linked chains of nodes does not allow for automatic garbage collection of nodes that have become unreachable to make their perfectly good memory bytes available for later node allocations. Therefore, once your program logic knows that some node `n` is no longer needed, it should call `IntList.release(n)` to release the entire chain of nodes starting from `n`. (To release only that one node instead of the entire chain, you need to first call `IntList.setNext(n, 0)`. Make sure to take the index of the first node of the rest of the chain for safekeeping to avoid the rest of that chain becoming an unreachable memory leak!)

For convenience, the `IntList` class already defines a `static` method `toString(int n)` to convert the chain starting from node `n` into a human-readable string representation. You should carefully read through this method until you understand how it works, followed by a similar study of the two other educational example methods `removeFirst(int n, int k)` and `reverse(int n)` also operating on chains, to serve as inspirational examples for the methods that you need to write in this project. You can run the `main` method of `IntList` to see a short demo of these methods in action.

The `IntList` class keeps internally track of which nodes have already been allocated and which are still available for future allocations. (The nodes that are still available are organized in a single chain of free nodes to make both node allocation and release work in O(1) time.) The `IntList` class also internally enforces **memory safety** so that once some node has been `released`, any attempt to read or modify its key or successor through the public interface of `IntList` will cause an `IllegalStateException` being thrown.

## Automated Tester

The instructors provide an automated tester class `IntListMethodsTest.java` that is used to check and grade your project. Used on the command line, it has the format

```
java IntListMethodsTest SEED ROUNDS SIZE VERBOSE
```

where `SEED` is the seed to initialize the pseudorandom number generator that produces the test cases, `ROUNDS` is the number of tests to run, `SIZE` is the number of elements in the test lists, and `VERBOSE` determines whether the tester should print out the intermediate results for your debugging purposes. Each test case consists of an array of integers that is first converted into a list.

Your code, using the methods defined below, will first `release` all the nodes whose key is divisible by a randomly chosen k between 2 and 7, and then sort the nodes that remain. An example run for five rounds with twenty elements each might look like the following:

```
matryximac:Java 305 ilkkakokkarinen$ java IntListMethodsTest 777 5 20 true
Original: [12, 19, -18, 2, 13, -10, -13, 11, -6, -4, 3, -4, -11, 13, 20, 5, -20, 1, 0, -9]
After removeIf(3): [19, 2, 13, -10, -13, 11, -4, -4, -11, 13, 20, 5, -20, 1]
After sort: [-20, -13, -11, -10, -4, -4, 1, 2, 5, 11, 13, 13, 19, 20]
Original: [-13, 19, 1, 3, -15, 11, -6, 16, -18, -18, -10, 4, -11, 2, -20, -6, 6, -4, 20, -19]
After removeIf(3): [-13, 19, 1, 11, 16, -10, 4, -11, 2, -20, -4, 20, -19]
After sort: [-20, -19, -13, -11, -10, -4, 1, 2, 4, 11, 16, 19, 20]
Original: [12, -4, -13, 19, 18, -2, -9, 12, -10, 5, 15, 20, -15, -10, -12, -2, -14, 7, -14,
17]
After removeIf(2): [-13, 19, -9, 5, 15, -15, 7, 17]
After sort: [-15, -13, -9, 5, 7, 15, 17, 19]
Original: [-19, -5, 3, 4, -3, -7, -9, -4, -15, -14, -5, -10, -7, -6, -6, 17, 6, -17, -15, -2]
After removeIf(6): [-19, -5, 3, 4, -3, -7, -9, -4, -15, -14, -5, -10, -7, 17, -17, -15, -2]
After sort: [-19, -17, -15, -15, -14, -10, -9, -7, -7, -5, -5, -4, -3, -2, 3, 4, 17]
Original: [-9, -6, 10, 5, 19, -14, 8, 15, 18, 6, 15, 3, 2, 9, 4, -16, -3, 0, -3, -8]
After removeIf(5): [-9, -6, 19, -14, 8, 18, 6, 3, 2, 9, 4, -16, -3, -3, -8]
After sort: [-16, -14, -9, -8, -6, -3, -3, 2, 3, 4, 6, 8, 9, 18, 19]
Method call counts: getKey 602, setKey 0, getNext 776, setNext 286.
3 123456789 Kokkarinen, Ilkka
```

The last line prints out the total time in milliseconds, followed by the author information. You can now easily generate new test cases small enough for you to eyeball for correctness simply by changing the random `SEED` on that command line. If your code fails the test, the reported running time will be absurdly large, and its exact value tells you the nature of the bug in your code:

| **Running time** | **The type of bug in your code** |
|---|---|
| 9999999 | Your `removeIfDivisible` method failed to work correctly. |
| 9999998 | Your `sort` method somehow produced a chain that contains different nodes than the original chain that was given to your method. |
| 9999997 | Your `sort` method failed to correctly sort the nodes in the right order of keys. |
| 9999996 | Some nodes became memory leaks and were lost. This means that either your `remove` or `sort` algorithm lost track of them without calling `release`. |
| 9999995 | During the sorting, you tried to either create a new node (that's a paddlin), use a nonexistent node (that's a paddlin), or reassign the key of some node (ooh, you better believe that's a paddlin). These nodes are for regular programming, not for fancy programming. |

If the command line argument `VERBOSE` is set to `false`, only this last report line gets printed. This allows us to run some pretty humongous tests without flooding the screen with endless outputs, for example the following instance of `[DrEvil]`*100 million*`[/DrEvil]` nodes (counted before calling `removeIfDivisible`) that turns out to take about 36 seconds on prof. Kokkarinen's trusty old Mac desktop:

```
matryximac:Java 305 ilkkakokkarinen$ java IntListMethodsTest 1234567 1 100000000 false
36754 123456789 Kokkarinen, Ilkka
```

# Grading

The project is graded for correctness and execution time. The grading TA will compile and test your project using a secret `SEED` that is the same for all submissions, with `ROUNDS` and `SIZE` set to equal to ten and one million, respectively. To receive any marks at all, **your project must correctly pass this test within the time limit of one minute**. If the execution of the automated tester has not terminated by that time, your project code is considered to be too slow (or even worse, has some bug that causes it to get stuck in an infinite loop), and will be rejected and receive a zero grade. Make sure to test your code with a large number of different seeds before submitting it. (For example, have the tester run overnight for at least a thousand rounds.)

The projects that pass this first big hurdle of working correctly are then sorted based on their running time. This list is then grouped into six rough portions. The projects in these six portions will receive a project mark of 10, 9, 8, 7, 6 and 5 points, respectively.

As an added incentive and reward, for whatever it is worth, prof. Kokkarinen will write a letter of recommendation to the student whose project submission is the fastest. The winner is also entitled to refer to him- or herself with the title of "The Fastest Gun West of Mississauga" for the duration of the Fall 2018 term. In case that several project submissions end up near the top with their running times being too close to call, this photo finish will be resolved with another test with the value of `SIZE` set to one hundred million to amplify every bit of inefficiency in your code.

# Required Methods

Your submission must consist of exactly one source code file `IntListMethods.java` and no other files. (Defining nested classes inside that class is acceptable.) However, in this particular project, **you are not allowed to call any methods other than those that are in the class `IntList` and that you write yourself in `IntListMethods`.** You are also **not allowed to import or use classes from any packages whatsoever** other than `java.lang`. All the required logic of linked lists and sorting must therefore be written by you from scratch!

Your class must have the following exact methods so that the automated tester can compile and run. How you choose to implement these methods is entirely up to you, within the constraints specified. Do something intelligent based on what you have learned in this course so far.

```
public static String getAuthorName()
```

Returns your name in the format `"Lastname, Firstname"` exactly as it appears on RAMMS.

```
public static String getRyersonID()
```

Returns your Ryerson student ID as a string without any spaces. For example, `"123456789"`.

```
public static int removeIfDivisible(int n, int k)
```

Given the first node `n` of a chain, traverse that chain and `release` all nodes for whose keys are divisible by `k`. Same as with all methods that mutate the contents of chains of nodes, this method should return the index of the first node of the resulting chain. This return value can even be 0 to denote an empty chain, if this operation happens to remove everything and leaves no nodes inside the chain.

```
public static int sort(int n)
```

Given the first node `n` of a chain, rearrange the nodes of that chain so that their keys are in sorted ascending order. Again, this method should return the index of the node that ends up being the first in this sorted chain.

Your sorting method must work so that **it does not create new nodes to represent the result**, but rearranges the existing nodes into the sorted order. (This requirement is sneakily enforced by the automated tester by initializing the `IntList` structure to allow only a total of `SIZE + 5` nodes to be created.)

Under that hard constraint, you get to freely choose the sorting algorithm that you implement, modified to operate on linked lists instead of the usual random access arrays. Due to the sheer size of the test cases used in project marking, any simple $O(n^2)$ sorting algorithms such as **insertion sort** should be out of the picture from the get-go. Of the $O(n \log n)$ family of sorting algorithms, **merge sort** or **quicksort** ought to be the easiest to modify to operate on linked lists (despite the impression that some textbooks like to present, neither one of these classic algorithms actually *needs* random access to elements to be able to operate), but again, you are allowed to implement any sorting algorithm you want. Once you get your chosen sorting algorithm to work correctly, try out different ways to shave down its running time. You are also allowed to exploit the fact that the integer keys to be sorted are random numbers drawn uniformly from the range { `-SIZE`, ... , `SIZE` }.

However, to eliminate the unsportsmanlike possibility of first copying the keys into a new integer array, followed by sorting that array with some good array sorting method copy-pasted from any one of the zillion possible online sources, and then coming back by copying the sorted keys from the array back into the list, the tester will internally **prevent the access** to the method `IntList.setKey` during the sorting so that **modifying the keys in nodes during sorting is not allowed.** This forces your sorting algorithm to actually rearrange the nodes of the list, instead of merely rewriting the sorted keys in the nodes without having to break down and reassemble the underlying structure of the chain. Which was the entire purpose of this project to demonstrate the flexibility of linked lists.