Arraylist Implementation

by

Brent Orlina

Computer Programming II

Nicholas Seward

Implementation

Java's Arraylist implementation is a straightforward concept.

1. It uses an array with a limited capacity to store its data.
2. It creates a new array with a bigger capacity if it hits the limit.

However, Java uses objects to store its data.

It is unable to store primitives such as `int` or `char`.

Instead, they use objects or boxed primitives such as `Integer` or `Character` to be able to handle a wide variety of data.

To simplify the re-implementation, the data type was constrained to only using `int`.

Arraylist Implementation

```
/* Java's Implementation */
private transient E[] data;

/* My Implementation */
private int[] vList;
```

Methods Implemented

The implementation uses an interface, `CapstoneList`, which include the following methods:

- `add(int index, int value)`
- `add(int value)`
- `contains(int value)`
- `get(int index)`
- `indexOf(int value)`
- `pop(int index)`
- `remove(int value)`
- `size()`

Arguably, the two most important methods are `add()` and `remove()`, which are the two methods that were later benchmarked.

CapstoneList Interface

```
public interface CapstoneList
{
    public void add(int index, int value);
    public void add(int value);
    public void clear();
    public CapstoneList clone();
    public boolean contains(int value);
    public boolean equals(CapstoneList list);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public int lastIndexOf(int value);
    public int pop(int index); // Removes specified index
    public boolean remove(int value); // Removes first occurence
    public int set(int index, int value);
    public int size();
    public CapstoneList subList(int fromIndex, int toIndex);
    public int[] toArray();
}
```

Pop / Remove Implementation

The `pop()` method was implemented in two different ways.

The naive method uses a `for-loop` to shift any subsequent element to the left by one index when the specified element is removed.

The other method, Java's implementation, uses `System.arraycopy()` to copy a portion of the data array to the left by one index.

This can also be applied to the `add()` method when inserting an element in a specified index.

However, this wasn't implemented as the benchmarking simply added elements to the right most index.

Pop / Remove Implementation

```java
if(index < 0 || index >= count) throw new IndexOutOfBoundsException();

int r = vList[index];

/* Naive Implementation */
for(int i = index; i < count-1; i++) {
        vList[i] = vList[i+1];
}

/* Java Implementation */
int move = count - index - 1;
if(move > 0) {
        System.arraycopy(vList, index + 1, vList, index, move);
}

count--;
return r;
```

Benchmarking Methods

The methods `add()` and `pop()` of both the re-implementation and Java's implementation were benchmarked and compared against each other.

The `add()` method was exponentially benchmarked from `2^0` to `2^29` as the number of elements added to the list. The elements were added the the right-most index of the list.

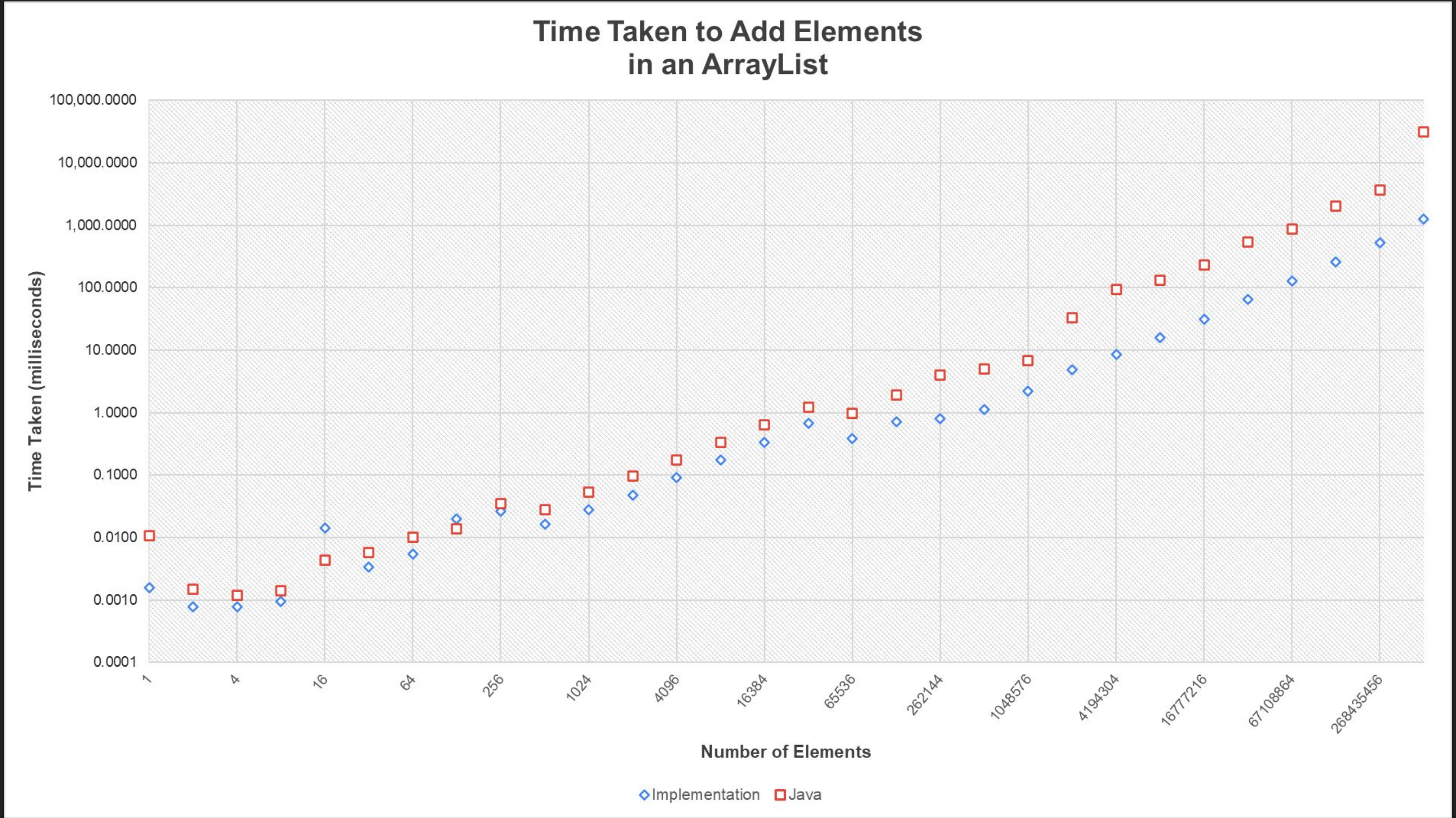
The `pop()` method was exponentially benchmarked from `2^0` to `2^19` as the number of elements removed to the list. The elements were removed from the `0th` index of the list.

Both the naive and Java implementations were benchmarked.

Pop / Remove Implementation

```java
public static long implementAdd(ArrayList list, int n) {
        long start = System.nanoTime();
        for(int i = 0; i < n; i++)
                list.add(i);
        long rTime = System.nanoTime()-start;
        return rTime;
}

public static long javaAdd(java.util.ArrayList<Integer> list, int n) {
        long start = System.nanoTime();

        for(int i = 0; i < n; i++)
                list.add(i);
        long rTime = System.nanoTime()-start;

        return rTime;
}

public static long implementPop(ArrayList list, int n) {
        for(int i = 0; i < n; i++) list.add(i);

        long start = System.nanoTime();
        for(int i = 0; i < n; i++)
                list.pop(0);
        long rTime = System.nanoTime()-start;

        return rTime;
}

public static long javaPop(java.util.ArrayList<Integer> list, int n) {
        for(int i = 0; i<n; i++) list.add(i);

        long start = System.nanoTime();
        for(int i = 0; i < n; i++)
                list.remove(0);
        long rTime = System.nanoTime()-start;

        return rTime;
}
```

.add() Data


Time Taken to Add Elements
in an ArrayList
Time Taken (milliseconds)
100,000.0000
10,000.0000
1,000.0000
100.0000
10.0000
1.0000
0.1000
0.0100
0.0010
0.0001
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
67108864
268435456
Number of Elements
Implementation
Java

.add() Results

The data shows that the re-implementation is slightly faster in smaller to medium number of elements added to the list.

As the number of elements grow, the gap between the re-implementation and Java's implementation stretch further with the re-implementation taking the lead.

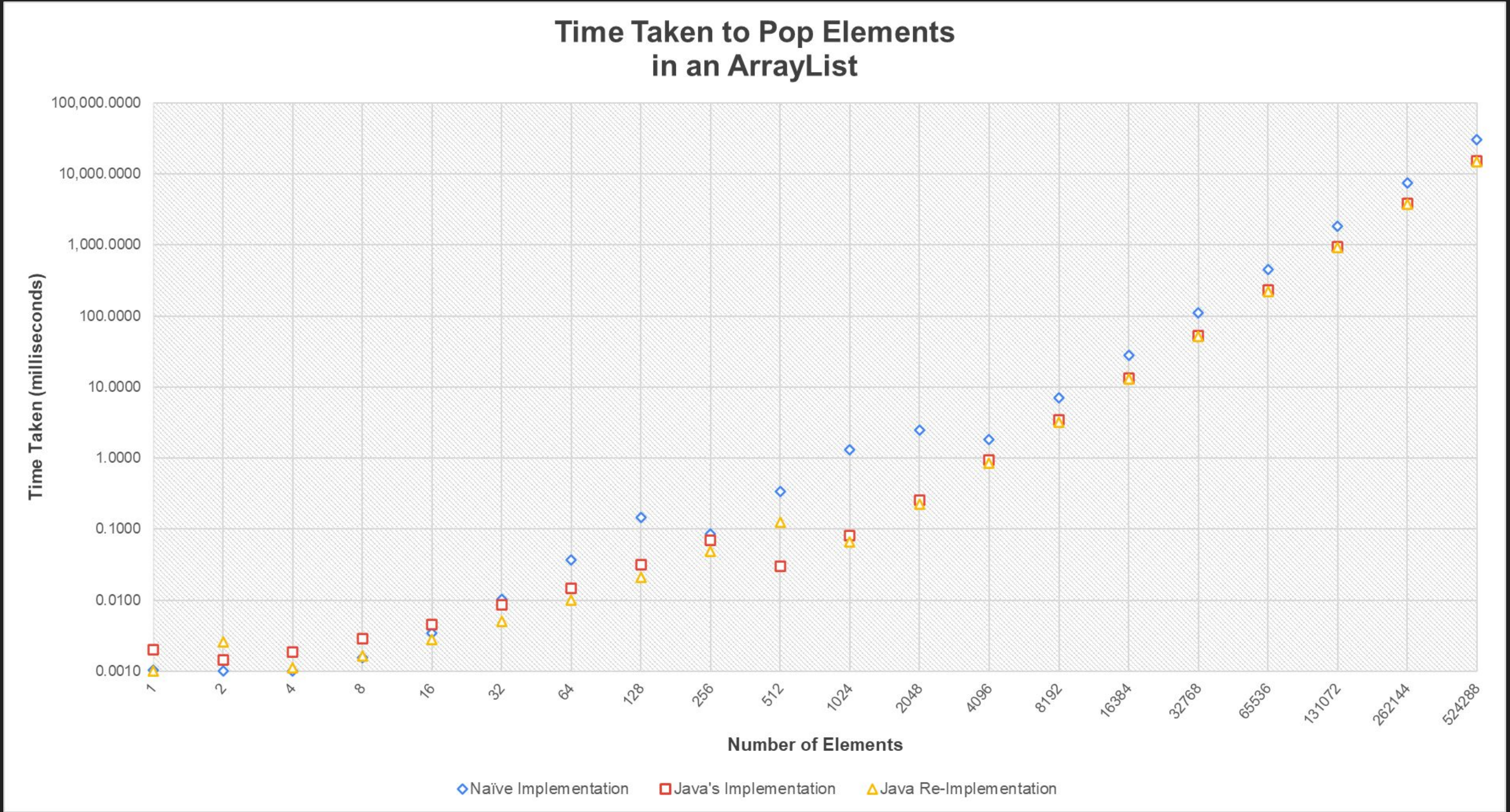
It is also interesting that only adding one element takes more time than adding eight or sixteen elements.

There is also an outlier at adding sixteen elements, with the time of the re-implementation jumping up to over 0.01 milliseconds.

.add() Data

N	Implementation	Java
1	0.001584	0.010625
2	0.000792	0.001500
4	0.000791	0.001208
8	0.000959	0.001417
16	0.014083	0.004375
32	0.003416	0.005708
64	0.005458	0.010084
128	0.020125	0.013791
256	0.026250	0.034750
512	0.016209	0.028042
1024	0.028042	0.053334
2048	0.048125	0.096334
4096	0.092084	0.175333
8192	0.173250	0.335208
16384	0.338292	0.631042
32768	0.682083	1.221500
65536	0.385167	0.966666
131072	0.707583	1.906167
262144	0.806583	3.965166
524288	1.125583	5.047750
1048576	2.196166	6.745583
2097152	4.901875	33.416208
4194304	8.594584	94.187542
8388608	15.830125	131.466000
16777216	31.380875	229.761708
33554432	64.813000	545.959083
67108864	126.352167	869.012125
134217728	260.033833	2032.858708
268435456	529.442250	3695.817417
536870912	1254.287167	30971.749625

.pop() Data


Time Taken to Pop Elements
in an ArrayList
100,000.0000
10,000.0000
1,000.0000
100.0000
10.0000
1.0000
0.1000
0.0100
0.0010
Time Taken (milliseconds)
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
Number of Elements
Naïve Implementation
Java's Implementation
Java Re-Implementation

.pop() Results

The data shows that all implementations perform similarly in smaller numbers of elements.

In the slightly bigger numbers of elements, it shows that the naive implementation is massively slower compared to the other implementations.

In huge numbers of elements, all three seem to perform and grow similarly. It is also important to note that there is still a significant gap between the naive implementation and the other two implementations as the scale of the graph is logarithmic.

.pop() Data

N	Java Re-implementation	Naive Implementation	Java
1	0.001042	0.001000	0.002000
2	0.001000	0.002584	0.001458
4	0.001000	0.001125	0.001875
8	0.001541	0.001625	0.002875
16	0.003416	0.002791	0.004542
32	0.010375	0.005000	0.008500
64	0.037166	0.010084	0.014833
128	0.144375	0.020959	0.031292
256	0.085833	0.048333	0.069917
512	0.336791	0.124792	0.030208
1024	1.324084	0.066792	0.080666
2048	2.460916	0.222916	0.256708
4096	1.832125	0.858417	0.939208
8192	7.105208	3.210334	3.487750
16384	27.753333	12.938417	13.357833
32768	109.985041	51.750458	53.459209
65536	455.965959	223.217167	231.433166
131072	1833.514792	916.736834	947.831208
262144	7360.304667	3702.146333	3801.086000
524288	30181.149000	14777.816042	15242.825125

Conclusion

In conclusion, the re-implementation's `add()` method performs faster than Java's implementation as the number of elements grows into larger numbers.

The `pop()` method for both implementations grow and perform similarly as the number of elements grow, although the naive implementation is still significantly slower.

A possible explanation as to why the re-implementation performs better or equal to Java's implementation is that it uses a primitive data type. Java used a boxed primitive type which is slower and inefficient. Primitive data types may simply be just more efficient.

Re-implementing the Arraylist could be useful if speed and performance is hugely important. However, the disadvantage is that it would take more time to re-implement and would be limited to only a single data type. If development speed is more important, it is better to just use Java's implementation.