Linked List Analysis

Matthew Galitz

University of Virginia, Charlottesville, VA 22904

Abstract.
The purpose of the experiment is to compare the runtime of various methods of the LinkedList data structure.

1 Introduction

1.1 Problem

The problem is to distinguish between the method of a LinkedList that are efficient and not very efficient. In this context, "efficient" describes how quickly a method is executed.

1.2 Description

The LinkedList data structure is a member of the Java Collections framework. Every element of the LinkedList is not stored in contiguous locations and are separate objects called a ListNode. Every ListNode stores a piece of data and has two pointers pointing to the previous and next ListNodes. ListNodes cannot be indexed directly. The LinkedList also utilizes a ListIterator object to keep track of one's place in a LinkedList and to assist with iterating throughout the structure. The LinkedList has the following methods:

1. size(): returns the number of ListNodes in the LinkedList
2. clear(): removes all ListNodes from the LinkedList
3. insertAtTail(): adds a ListNode to the end of the LinkedList
4. insertAtHead(): adds a ListNode to the beginning of the LinkedList
5. insertAt(): adds a ListNode at a specified index
6. insert(): adds a ListNode at location of some ListIterator
7. removeAtTail(): removes ListNode from the end of the LinkedList
8. removeAtHead(): removes ListNode from the beginning of the LinkedList
9. remove(): removes ListNode at location of specified ListIterator
10. find(): returns index of first instance of ListNode containing specified data
11. get(): returns data stored in ListNode at specified index

2 Methods

Each method was executed 100,000 times on a LinkedList of size 10,000. The LinkedLists contain random integers from 0 to 1,000. The amount of time it takes to complete 100,000 executions of each method is recorded in milliseconds. This process is repeated for 8 trials.

Number of Trials Run on each Method

Methods	Number of Trials
insertAtTail()	8
insertAtHead()	8
insertAt()	8
insert()	8
removeAtTail()	8
removeAtHead()	8
remove()	8
find()	8
get()	8

The average and standard deviation of the data for each method are calculated using the following equations respectively.

$$A = \frac{\sum_{n=1}^{N} x_n}{N}$$

Where x_n is the time elapsed for trial n, N is the total number of trials, and A is the average time for a particular method.

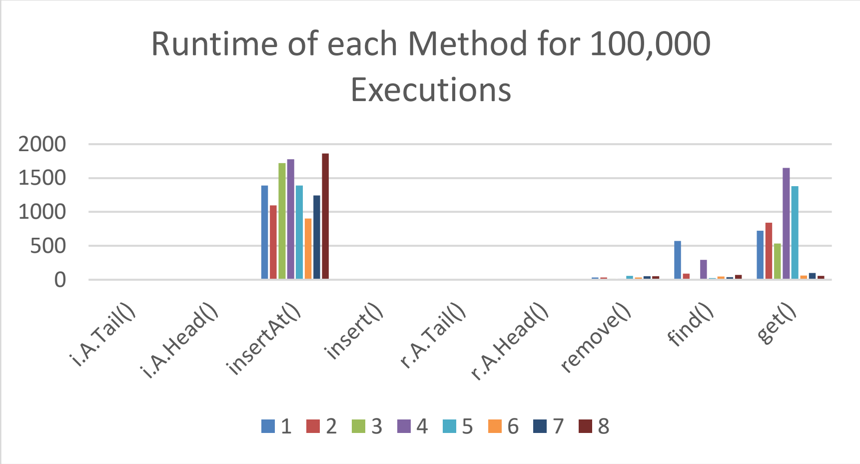
$$s = \sqrt{\frac{\sum_{n=1}^{N}(A - x_n)^2}{N - 1}}$$

Where A is the average time elapsed for a particular method, x_n is the time elapsed for trial n, N is the total number of trials, and s is the standard deviation of a particular method.

3 Results

Runtime of each Method for 100,000 Executions

	1	2	3	4	5	6	7	8	Avg.	Std.
i.A.Tail()	7	7	7	7	6	7	6	6	6.63	0.52
i.A.Head()	7	4	4	5	5	6	6	3	5	1.22
insertAt()	1388	1096	1720	1776	1388	903	1245	1861	1422.13	341.45
insert()	6	7	7	6	6	6	6	5	6.13	0.64
r.A.Tail()	1	1	2	1	1	1	1	1	1.13	0.35
r.A.Head()	1	1	4	1	0	1	0	0	1	1.22
remove()	35	34	16	13	58	34	53	50	36.63	16.49
find()	571	92	9	293	24	47	37	72	143.13	194.9
get()	725	840	535	1649	1382	62	100	55	668.5	608.08



4 Conclusion

The slow methods included insertAt(), find(), and get(). The fast methods included insertAtTail(), insertAtHead(), insert(), removeAtTail(), removeAtHead(), and remove().

This is not surprising because the slower methods used for-loops. Iteration drastically increases the runtime of these methods. The faster methods did not use for-loops. These observations agree with the theoretical runtimes of these methods. The slower methods have a time complexity of O(n) and the faster methods have a time complexity of O(1). This explains the differences in runtimes.

Starting at a size of 75,000-100,000 elements, the methods experienced a significant increase in runtime.