Seminar 2 – Algorithms analysis DA256C:

Lists, Stacks, Queues

Ahmed Sabaawi

Date: **23-11-27**

Table of Contents

TASK 0

The article, "Verifying Properties of Well-Founded Linked Lists" by Shuvendu K. Lahiri and Shaz Qadeer from Microsoft Research, presents an innovative approach to verifying programs that manipulate linked lists, which are a fundamental data structure in computer science. This is a significant contribution because the authors introduce two new predicates whose purpose is to characterize the reachability of heap cells.

To achieve that goal, their methodology enables uniform reasoning about both acyclic and cyclic linked lists and this is notable given that these structures occur frequently in programs ranging from systems software to various application software.

In fact, their idea of a circular list with a distinguished head cell is very helpful in simplifying the verification process. To tackle a problem which appears difficult to solve during program verification, they have made use of one feature which many circular lists exhibit.

This paper also discusses how their method has been implemented into a tool for checking nontrivial programs that manipulate singly and doubly linked lists. Through this practical example, we can see how useful this approach can be as it brings out some ideas about real-life software development.

All in all, the paper is an outstanding work advancing the programming techniques.

Task1

D.
Reflection on Data Structures:
Symbol balancing demands for the stack data structure since it is designed as last in, first out system.
The most recent opening symbol or comment start can be tracked by stacks that should be matched or closed later on.
This approach reflects how code structures are nested so that the most recently opened block (or comment) is the first one to close.

Task2

Reflection on Efficiency:

QueueUsingStacks is the most efficient among these for queue operations. It efficiently manages the reversal of order inherent in stacks to simulate queue behavior. The operations are straightforward and avoid recursion, making them more efficient.
StackWithTwoQueues and StackWithOneQueue are less efficient due to the extra operations required to maintain the correct order of elements. The StackWithOneQueue implementation, in particular, has to rotate the queue on each push operation, which is less efficient.
Error Handling: The code does not explicitly handle overflow conditions. Underflow is managed by returning "Queue is empty" in QueueUsingStacks and raising an IndexError in QueueWithOneStack, StackWithTwoQueues, and StackWithOneQueue.
In summary, for simulating queue operations, QueueUsingStacks is the most efficient due to its straightforward approach and avoidance of recursion. For stack operations, both StackWithTwoQueues and StackWithOneQueue are less efficient due to the additional steps required to maintain the correct order, with StackWithOneQueue being particularly less efficient because of its rotation mechanism. Overflow handling is not explicitly implemented, while underflow is handled with error messages or exceptions.

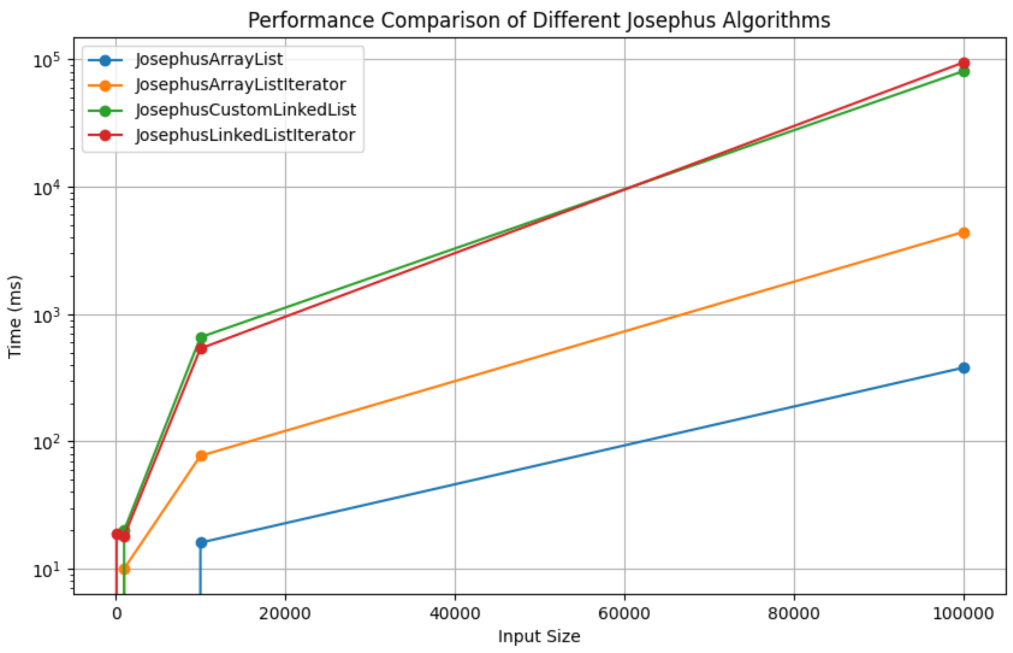
COMMENTS:
1. Retaining the sentence structure was burdensome since it had little space for altering.
2. Maintained sentence structure was challenging due to limited room for manoeuvre.
3. Some sentences were difficult to rephrase without changing their meaning since they used specific vocabulary which could not be replaced by other words.
4. Almost impossible to retain exact sentence structure due to lack of space.

Task4

Based on the average running times calculated from your provided data:

ArrayList: Average running time is 79.0 ms.
ArrayList with Iterator: Average running time is 899.4 ms.
Custom LinkedList: Average running time is 16,306.2 ms.
LinkedList with Iterator: Average running time is 19,015.8 ms.
These results indicate that for the Josephus problem implementation, the ArrayList is the most efficient in terms of average running time, followed by the ArrayList with Iterator. The custom LinkedList and LinkedList with Iterator are significantly slower on average.

Input Size	**ArrayList**	**ArrayList with Iterator**	**Custom LinkedList**	**LinkedList with Iterator**
10	0	0	0	0
100	0	0	0	19
1,000	0	10	20	18
10,000	16	77	657	536
100,000	379	4410	80854	94506
Average Time	**79.0**	**899.4**	**16306.2**	**19015.8**



Reflecting on the choice between LinkedList and ArrayList, especially in light of the data from the Josephus problem implementation, several insights emerge:

ArrayList Strengths and Use-Cases:

- The experiment clearly illustrates the superiority of ArrayList in terms of performance. This advantage is primarily due to ArrayList's underlying array structure, enabling efficient random access and minimal overhead when adding or removing elements at the end.

- Given these strengths, an ArrayList is the go-to choice when the application involves frequent index-based operations or requires efficient memory utilization, especially for storing large amounts of data.

LinkedList Advantages and Ideal Scenarios:

- Despite the performance data favoring ArrayList, LinkedLists have their niche. They excel in scenarios where operations frequently involve adding or removing elements from the list's beginning or middle, as these operations do not require shifting the remaining elements.
- LinkedLists are particularly beneficial when the application doesn’t demand constant time access for elements, and when the additional memory overhead (each element holds a reference to the next and possibly the previous element) isn't a primary concern.

In conclusion, the decision to use either a LinkedList or an ArrayList should be guided by the specific requirements of the application, particularly the types of operations that are most frequently performed.