Kristianstad University
SE-291 88 Kristianstad
+46 44-250 30 00
www.hkr.se

DA256D, 7,5 credits
Semester Year e.g. Autumn Semester 2021
Faculty of Natural Science

Seminar 2 – Lists, stacks and queues

Sandra Kaljula

Author

Sandra Kaljula

Title

Lists, stacks and queues – Seminar 2

Supervisor

Kamilla Klonowska

Examiner

Kamilla Klonowska

Abstract

Data structures like lists, stacks and queues have different operations and time complexities for each of them. In this report these three data structures operations are compared by mimicking one another’s operations.

Keywords

Algorithms, data structures, big-O, lists, stacks, queues.

Content

Introduction

This report is about solving the seminar 2 tasks and some reflections on them.

Limitations:

The algorithms are not guaranteed to be the most efficient ones out there, since they were the first working ones I came up with or were found on the internet. This is due to the time limit for this exercise. The exercises are also coded and benchmarked in java, which might bring different running times compared to other programming languages.

Method

First, I chose to use java to program my algorithms in, because I have lately used both python and Java and I would want to be better at this language.

Second, a file structure was created with all the files needed for execution. It is very important to have everything organized, since there were over 10 files in total.

Third, the information needed on lists, stacks and queues was gathered.

Fourth, I moved into writing the algorithms. It was decided to borrow some of the algorithms from the internet or the course literature and altered them to understand them. The references are found in the references part of the report as well as in the java files.

Then finally at the end of task 4 the tests were run automatically on that task on a computer with AMD Ryzen 3 1300X Quad-Core Processor 3.80 GHz. The results were drawn in excel as graphs and tables which are attached with this report.

Results

Task 0

Reflection on Verifying Properties of Well-Founded Linked Lists.

This article is written on a very high level considering the authors are from Microsoft. In this article they represent their way of verifying the linked data structures reachability with the focus on the head cells. It is for all the circularly linked as well as not circularly linked single- and double linked lists. Their program simply verifies the correctness of the programs that use linked lists. This article goes in deep detail about the different linked list methods that can be used on the linked list heads - like remove() and add(), describing exactly the behind the scenes of these approaches. Like the way they are mapped and what kind of datatypes are used and in which ways[1].

I feel like this article is very hard for me to understand fully personally because there are so many new words and approaches used. Some of the text I understand since before we have gone through it in the mathematics course when it comes to the calculations, but also the operative system course since they write about how the cells are allocated and how the processes work.

Task 1

For task1 there is a menu in which the user can input the filename and language. In the actual logic part of the program, the file is read line by line and for every line the characters in the line are set in a queue. This makes it possible to check each character for themselves. It also helps with the order in how the balancing symbols come. With the queue having the peek method it was possible to peek at the next element and see if it is an important one. An example of that would be when both the opening and closing symbols were 2 characters long /* and */. This made my solution of N in the power of 2 complexity, because of the 2 nested while loops.

I think the most suitable data structure in this case is a stack, because with a stack it is only possible to push to the top and pop from the top of the stack. When coding correctly with balancing symbols each opening symbol should have a closing symbol and there should not be a closing symbol directly after another opening symbol. Another thing that is not a limitation is that peek(), clear() and poll() in a queue and pop(), push() and peek() in a stack are all of O(1) complexity. The only thing that might add more complexity to it would be going through the list of opening and closing symbols. But since these arrays are fixed size the time it takes will mostly be linear.

Task 2

As stated in the method part some of the code was borrowed or coded off inspiration from the examples on the internet. It was attempted to come up with the solutions and the logic behind the algorithms. The stack overflow messages are displayed once the size of the stack is way too much for the program to handle, but in this case the overflow message will never be reached since JVM automatically increases the limit and considers the programmer to know what is best. Underflow is when the queue or stack is empty and there are no elements to retrieve, but it's tried. For underflow I decided to take care of the exception with an if statement that checks if the queue or stack is empty, but for overflow for stacks I used the try-catch with exceptions.

I really wanted to see how Array Deque would work with this scenario, so I used it and I noticed soon after that it makes the last approach the most effective of them all. I also looked it up even further and found out that most people on stack overflow are using deque over a stack.

Considering some of the code was not coded the best way possible there might be inconsistencies in time complexity of the algorithms.

If the loops are not nested the time complexities are added together.

a)Implementation of a queue using two stacks.

<u>enqueue</u>

- for this I put everything from the first stack into the second stack O(N)
- pushed in the single number that was to be “queued” O(1)
- and lastly put everything back into the first stack O(N)

Time complexity: O(N)

<u>dequeue</u>

- Takes the first number from the stack and returns it O(1)

Time complexity: O(1)

b)Implementation of a queue using only one stack.

<u>enqueue</u>

- I push the single number into the stack O(1)

Time complexity: O(1)

<u>dequeue</u>

- I popped all the elements out of the single stack and stored them in an array O(N)
- I did not include the last one and stored it in a variable O(1)
- I pushed the elements back into the stack O(N)

Time complexity: O(N)

c)Implementation of a stack using two queues.

<u>push</u>

- Adding the bottom element to the second queue O(1)
- Adding the first queue to the second queue O(N)
- Swapping the names of the queues O(1)

Time complexity: O(N)

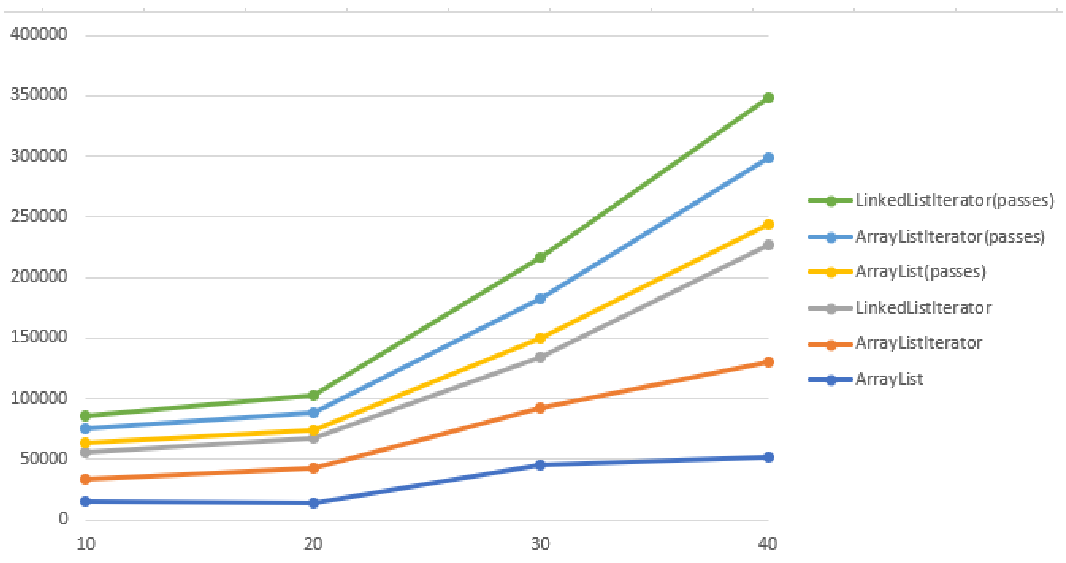
<u>pop</u>

- Just removing the last element O(1)

Time complexity: O(1)

d)Implementation of a stack using a single queue.

Array Deque



class was used, because it makes it possible to add elements to both ends.

<u>push</u>

- adds the single element to the start of the queue O(1)

Time complexity: O(1)

<u>pop</u>

- removes the first element in the queue O(1)

Time complexity: O(1)

Task 3

For this task I created two classes - Node and LinkedList. Inside the node I define the name, address and the object of the next node. There is also a method that returns a string with the name and address. For the LinkedList I decided to create add(), remove(), getByIndex() and printAll() methods. The add method simply adds another node to the linked list and removes the last element that was put into the linked list. GetByIndex() gets the value of the input index and printAll() uses the method in the Node class to print the name and address for each existing Node.

For this task I created 2 possible ways for doing it. First, I notice that using an array List to store all the nodes would be much faster since it is random access. But then I created a LinkedList class that works exactly as the LinkedList class in java, because it was the real task here.

Task 4

Figure 1 and table 1 showcase the running times for the three algorithms in nanoseconds. It compares the size from 10 to 40 in relation to passes made.

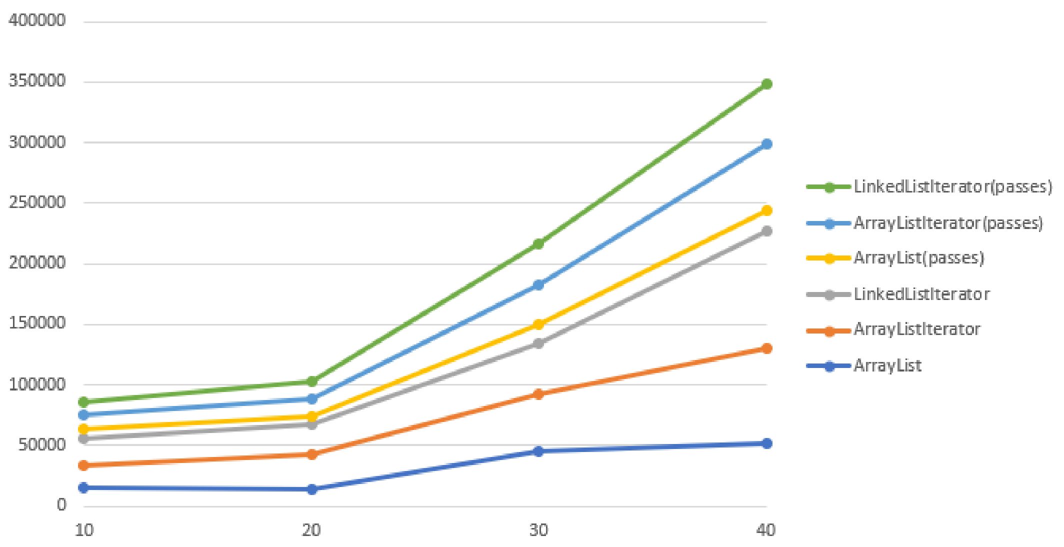


Figure 1 Time in nanoseconds for different input sizes(and passes) in relation to each other.

		Time in nanoseconds for algorithm:		
Passes	Size	ArrayList	ArrayList (I)	LinkedList (I)
2	10	15100	18800	21900
2	20	13200	29600	24400
2	30	45800	46600	41200
2	40	51700	78200	97100
1	10	7800	11200	11400
2	20	6900	14700	13900
3	30	14600	33200	33500
4	40	16700	55700	49400

Table 1 Runtime in nanoseconds for different input sizes and passes for the algorithms I in the parenthesis stands for "with iterator",

The array list and linked list approaches compared side by side tells us that the array list is more stable and less time consuming. See figure 1. This could be due to it being random access, whilst the linked list is sequential access and must go through every node prior to the node to find the desired one.

Comparing the algorithm that uses an array list and a list iterator to a linked list that uses a linked list iterator, it is also clear that the array list in that case is also the most time efficient choice. But in this case the difference in times is not that big. See figure 2.

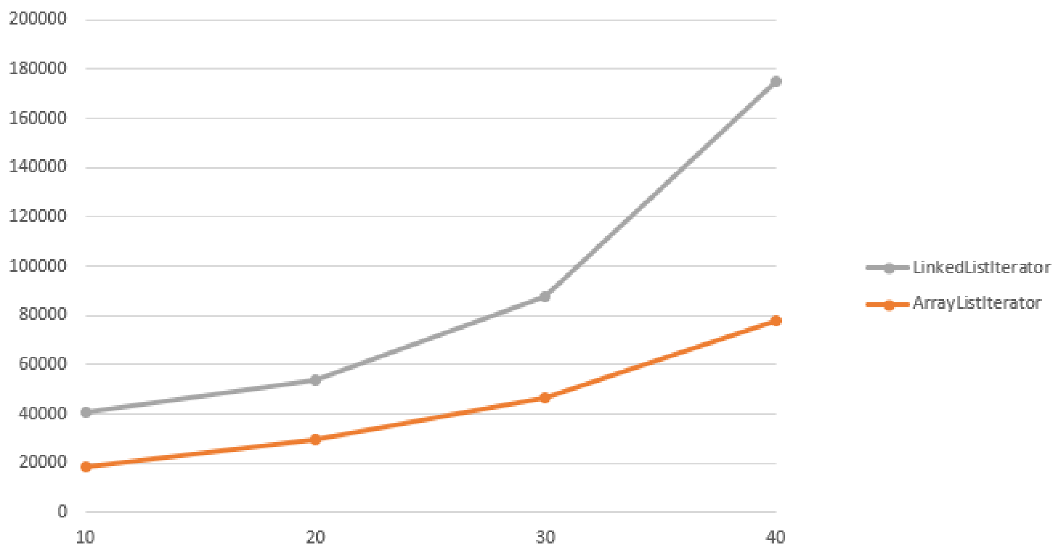


Figure 2 ArrayListIterator and LinkedList with Iterator side by side.

Meanwhile the list iterator adds extra time. At least when using it on an array list. Seen in Figure 3.

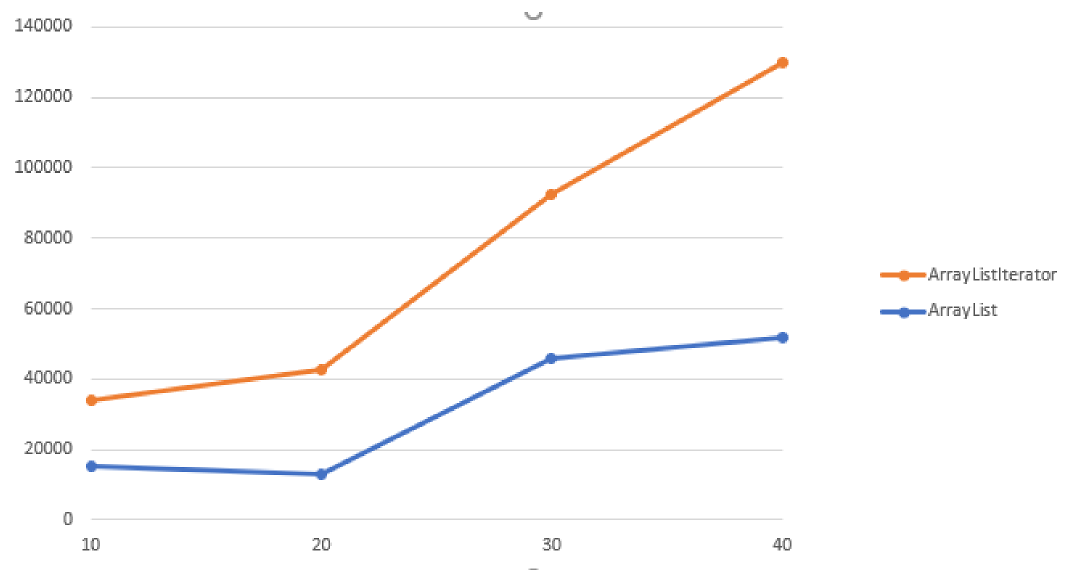


Figure 3 Running times for the ArrayListIterator and ArrayList approach in nanoseconds.

Another point that is very important in this game is how the number of passes between the actors affects the running time of this program. In figure 4 it is apparent that the more passes there are the longer time it is going to take to run. But when only looking at the algorithms with different passes we see that the graphs start going further away from each other with larger inputs. For smaller inputs the difference is less.

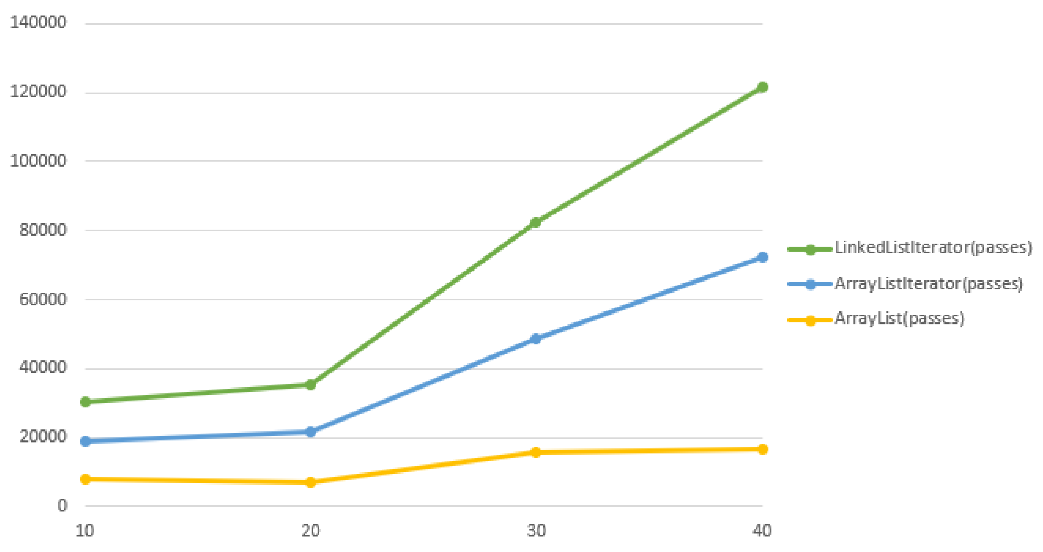


Figure 4 The three algorithms with increasing number of passes running time in nanoseconds.

There cannot be an exact formula for this case since there are two really important factors which are the passes and size. There are also other factors that affect the result. However, there is probably a way to make a formula for the approximation.

The results that were retrieved from my program I would say that the array list is overall the best option for this kind of a program. But when using an iterator together with a linked list the linked list might work even better for bigger data.

Conclusion

To summarize, this seminar was very interesting. There were many ways to get to learn about the different data structures, how they work and how to create one if we are ever missing one another in another programming language. I personally learned further on the time complexities for using different data structures. For the last tasks I could have probably used some larger input values as well, since discussing with my classmates it was discovered that LinkedList was much faster with larger values.

References

[1] Qadeer S, Lahiri S. Verifying Properties of Well-Founded Linked-Lists. 2006 January; p. 115-116. https://hkr.instructure.com/courses/4156/files/837226?module_item_id=216858

The code(also mentioned in each java file):

Task1:

Weiss, Mark. A. (2012), Data structures and algorithm analysis in Java. 3rd edition Harlow, Essex : Pearson. p 84.

Task2:

GeeksforGeeks. Queue using stacks [Internet].GeeksforGeeks: [cited date 2021-12-01]. Available from: https://www.geeksforgeeks.org/queue-using-stacks/

GeeksforGeeks. Implement stack using queue [Internet].GeeksforGeeks: [cited date 2021-12-01]. Available from: https://www.geeksforgeeks.org/implement-stack-using-queue/

Task4:

Priya. How to implement a program for Josephus problem in Java [Internet].[cited date: 2021-12-01]. Available from: https://titanwolf.org/Network/Articles/Article?AID=400f1234-15c8-4200-ac1e-3e6a0157bf7d