Append Operation Efficiency on Linked Lists and Arrays

Alexander Alvarez

Fall 2024

Introduction

The time efficiency of the append operation is to be measured on linked lists and arrays. A stack data structure will also be implemented using the linked list data structure. The stack will be benchmarked and it's implementation will be discussed.

The main point is to understand the linked list data structure by performing these benchmarks. Focus lies therefore on how the time changes when lists vary in size, i.e. the time complexity.

The general behaviours of these data structures are described as well as how they were created and implemented for benchmarking.

Method

The benchmarks will be performed using 10 sizes `n` starting from 100 and doubling up to 51200 (`n =` {100, 200,..., 25400, 51200}).

The append operation will be looped a certain amount of times (`loop`) where the average time for one operation is the total runtime divided by `loop`. The `loop` variable is set to `1000` for all benchmarks. Due to consistency, only the minimum time out of `10` runs for each `n` will be measured.

The function for measuring the time runs the append method in a `for` loop. The lists, arrays and stacks are declared within the append method in each iteration. The time is taken right before and after performing the append operation in that method. The time difference then returns and accumulates in a variable `t` in the `for` loop which becomes the total runtime.

```
private static long linkedBench(int n, int loop){
  long t = 0;
    for(int i = 0; i < loop; i++)
      t += appendLinked(n, loop);
  return t;
}
```

The benchmark methods are also run for a sufficient time before the actual benchmarks to enable the just-in-time compiler to make optimizations.

Linked list

The linked list data structure is a linear series of nodes connected by references. Data is stored in each node and each node points to the next node by a reference.

A linked list is made by creating a sub class for a node. The data and reference variables are then declared within this class for which the node then can be declared. The first node in the list is called the head and the last node is referenced to as tail. The head node is used to access the list and the tail node points to `null` to signify the end of the list.

The linked list class consists of several methods for managing the list with the append method being one such method. The append operation simply connects the head of one list (`b`) to the tail of another list (`a`). This is done by cycling to the tail of `a` and referencing it to the head of `b`.

```
public void append(LinkedList b){
  Node a = this.head; // get a
  while(a.tail != null) a = a.tail; // gycle to tail
  a.tail = b.head; // connect tail of a to head of b
}
```

As explained in the Method section, the append method for benchmarking is run in a `for` loop. The looped method creates list `a` with size `n` and list `b` with size `loop`. The method returns the time for one append operation.

```
public static long appendLinked(int n, int loop){
  LinkedList a = new LinkedList(n); // declare a
  LinkedList b = new LinkedList(loop); // declare b
  long t0 = System.nanoTime(); // time 1
  a.append(b); // append operation
  long t1 = System.nanoTime(); // time 2
  return (t1 - t0); // return append operation time
}
```

List `a` is run with the increasing size of `n` whilst list `b` has the fixed size `loop` in all runs. To measure the other way around where `a` is fixed and `b` increases, just switch `a` and `b` with each other in the append operation such that `b.append(a)`. Running the benchmark for linked list yields the result in table 1.

Runtime^1 increases linearly with the size of `n` which makes sense since the amount of nodes in `a` that the program cycles through to get to the tail is increasing by `n`. Runtime^1 therefore has the time complexity $O(n)$ whereas

size(n)	**runtime**^{1}	**runtime**$^1/(n)$	**runtime**^{2}	**runtime**$^2/(1)$
100	$0.12\mu s$	1.2	$1.0\mu s$	1.0
200	$0.24\mu s$	1.2	$1.0\mu s$	1.0
400	$0.44\mu s$	1.1	$1.0\mu s$	1.0
800	$0.85\mu s$	1.1	$1.0\mu s$	1.0
1600	$1.7\mu s$	1.0	$1.1\mu s$	1.1
3200	$3.3\mu s$	1.0	$1.0\mu s$	1.0
6400	$6.5\mu s$	1.0	$1.1\mu s$	1.1
12800	$13\mu s$	1.1	$1.1\mu s$	1.1
25400	$26\mu s$	1.1	$1.1\mu s$	1.1
51200	$57\mu s$	1.1	$1.1\mu s$	1.1

Table 1: Runtime and ratio where runtime^1 is list `a` appended to list `b` and runtime^2 is the other way around.

runtime^2 has the time complexity $O(1)$ due to the fixed amount of nodes in `b`. Dividing each runtime by their respective time complexity gives the ratio which for both cases is roughly 1.0. This shows that the time complexities most likely holds true for their respective case.

Array

Two different implementations were used to test the append operation on arrays. The first simply copies the contents of both array lists and creates a single array of size `n + loop` with the contents both lists in the correct order. The performance of this method is displayed in table 2.

size(n)	**runtime**^{1}	**runtime**$^1/(x)$	**runtime**^{2}	**runtime**$^2/(x)$
100	$0.71\mu s$	0.65	$0.62\mu s$	0.56
200	$0.78\mu s$	0.65	$0.69\mu s$	0.57
400	$0.82\mu s$	0.58	$0.92\mu s$	0.66
800	$1.0\mu s$	0.56	$1.2\mu s$	0.66
1600	$1.4\mu s$	0.53	$1.7\mu s$	0.67
3200	$2.7\mu s$	0.64	$2.8\mu s$	0.66
6400	$4.3\mu s$	0.59	$5.0\mu s$	0.68
12800	$7.0\mu s$	0.50	$8.9\mu s$	0.64
25400	$13\mu s$	0.50	$17\mu s$	0.66
51200	$30\mu s$	0.58	$37\mu s$	0.71

Table 2: Runtime and ratio where runtime^1 is list `a` appended to list `b` and runtime^2 is the other way around. Time complexity x stands for `n + loop`.

The problem with the first implementation is that the operation copies the same amount regardless of which list is appended to the other. This makes the runtime of both cases have the same time complexity of $O(n)$.

The second implementation utilizes the stack data structure for the arrays. The code for the stack data structure was taken and used from a previous assignment. The append operation is a `for` loop that for each iteration pushes an element from the stack array of `b` to the stack `a`.

```
public static long appendArrayStack(int n, int loop){
  DynamicStack a = new DynamicStack(); // declare a
  for(int i = 0; i < n; i++) a.push(i); // push values to a
  StaticStack b = new StaticStack(loop + n); // declare b
  for(int i = 0; i < loop; i++) b.push(i); // push values to b
  long t0 = System.nanoTime(); // time 1
  for(int i = 0; i < b.top; i++) a.push(b.stack[i]); // append operation
  long t1 = System.nanoTime(); // time 2
  return (t1 - t0); // return append operation time
}
```

The performance of the second method is shown in table 3.

size(n)	**runtime**^{1}	**runtime**$^1/(1)$	**runtime**^{2}	**runtime**$^2/(n)$
100	$3.9\mu s$	3.9	$0.067\mu s$	0.67
200	$3.9\mu s$	3.9	$0.11\mu s$	0.56
400	$3.8\mu s$	3.8	$0.20\mu s$	0.52
800	$3.4\mu s$	3.4	$0.38\mu s$	0.47
1600	$4.4\mu s$	4.4	$0.73\mu s$	0.46
3200	$6.8\mu s$	6.8	$1.4\mu s$	0.45
6400	$2.5\mu s$	2.5	$2.8\mu s$	0.44
12800	$2.5\mu s$	2.5	$5.6\mu s$	0.44
25400	$2.5\mu s$	2.5	$11\mu s$	0.44
51200	$2.5\mu s$	2.5	$23\mu s$	0.44

Table 3: Runtime and ratio where runtime^1 is list `a` appended to list `b` and runtime^2 is the other way around.

The second method displays more efficicent results than the first method. For runtime^1, the program only needs to push the fixed amount of elements in `b` since the top pointer is at the end or "tail" of stack `a`. This gives a time complexity of $O(1)$.

The top pointer is also the reason that the program pushes the elements in order of the array from stack `b`. Otherwise, using the `pop()` operation would connect the tails of both stacks to each other.

The program pushes the increasing amount of elements `n` in runtime^2 which gives the second case the time complexity $O(n)$.

Stack

The stack data structure can utilize much of the linked list data structure for implementation. A stack could be implemented by using the "head" node as the "top" of the stack and having every "next" node being the "previous" top node.

The push() operation would simply put the current top node as the previous node and make a new top node for the pushed value.

```
public void push(int value){
  top = new Stack(value, top); // makes new top node
}
```

The pop() operation returns the value of the current top node and makes the previous node the new top node.

```
public int pop(){
  int val = top.current; // get value
  Stack popped = top.previous; // previous node
  if(popped == null) top = null; // if previous is null
  else top = new Stack(popped.current, popped.previous); // new top node
  return val; // return value
}
```

The stack data structure is now implemented using linked lists. The append operation looks basically identical to the one in linked list. The resulting time performances are displayed in table 4.

size(n)	**runtime^1**	**runtime^1/(n)**	**runtime^2**	**runtime^2/(1)**
100	0.12μs	1.2	1.0μs	1.0
200	0.24μs	1.2	1.0μs	1.0
400	0.44μs	1.1	1.0μs	1.0
800	0.85μs	1.1	1.0μs	1.0
1600	1.7μs	1.1	1.0μs	1.0
3200	3.3μs	1.0	1.0μs	1.0
6400	6.6μs	1.0	1.0μs	1.0
12800	13μs	1.0	1.0μs	1.0
25400	28μs	1.1	1.0μs	1.0
51200	57μs	1.1	1.0μs	1.0

Table 4: Runtime and ratio where runtime^1 is list a appended to list b and runtime^2 is the other way around.

The performances shown in table 4 looks almost identical to those in table 1 which makes sense since this stack is basically a linked list.

Discussion

The similarities between linked lists and arrays is that both are linear data structures that stores data. The differences are how they work and how the data is handeled.

The time complexity of the append operation for arrays can be made to be similar to that of linked lists by utilizing stacks. If done right, both can have the same complexity, although the actual times might differ a lot.

What seperates both data structures is that linked lists is non-contiguous, has sequential access and is more efficicent at insertion and deletion. Part of the memory is allocated to an array which leaves some parts in the memory unused if the array isn't full. Linked lists only uses enough memory for what it needs which makes it a bit slower. Linked lists are more efficicent at inserting and deleting data due to its dynamic structure compared to the fixed sizes of arrays.