Stephen Belden

Chris Ruiz

Meghan Haukaas

2015-Sept-16

**Self-Adjusting List Tests**

**Description of Test Cases**

There are two cases for valid non-trivial inputs:

* Test Case 1 is the perfect case, in which all of the elements searched for are in the list. We accomplish this by setting the List Length and Max Value to the same number.
* Test Case 2 is the case in which at least 1 number searched for is not in the list. When a number is not in the list, every element of the list must be visited, regardless of whether the self-adjusting algorithm is used. Because of this, searching for an element that is not in the list always takes N steps, were N is the length of the list. In these tests we exaggerate this problem by setting the Max Value to ten times the List Length.

Several cases are not tested:

* There is a trivial case when the List Length is 0. In this case no elements are ever compared, regardless of what values are searched for. The Max Value and Cluster Size can be any value, and this case will still be trivial, with a total of 0 steps.
* Setting the Max Value to a number less than the List Length is an invalid set of inputs, as every element of the list must be unique.
* No negative inputs are considered valid.

**Test Case Construction**

Each test case was tested with these values:

* Cluster size of 1, 7, 30
  + A value of 1 represents a linear search that visits each element once, in order
  + A value of 7 is an average use-case
  + A value of 30 tests how the self-adjusting list scales with large clusters, and how the container reacts when the cluster size is larger than the list length.
* List Length of 10, 100, 1000

**Expected Outcomes**

Test Case 1 – Perfect Case

|  |  |  |  |
| --- | --- | --- | --- |
|  | **List Size 10** | **List Size 100** | **List Size 1000** |
| **Cluster size 1** | self-adjusting larger | larger | larger |
| **Cluster size 7** | smaller | smaller | smaller |
| **Cluster size 30** | larger | smaller | smaller |

Test Case 2 – More Misses than Hits

|  |  |  |  |
| --- | --- | --- | --- |
|  | **List Size 10** | **List Size 100** | **List Size 1000** |
| **Cluster size 1** | Self-adjusting larger | larger | Larger |
| **Cluster size 7** | smaller | smaller | smaller |
| **Cluster size 30** | smaller | smaller | smaller |

**Actual Outcomes**

Measured in steps/find call, lower is faster.

Green cells matched our predictions, red cells did not.

Format: without self-adjusting, with self-adjusting

Test Case 1 – Perfect Case

|  |  |  |  |
| --- | --- | --- | --- |
|  | **List Size 10** | **List Size 100** | **List Size 1000** |
| **Cluster size 1** | 5.5,7.7 | 50.5, 73.49 | 500.5, 741.793 |
| **Cluster size 7** | 10.0408, 9.83673 | 53.2504, 19.3918 | 502.874, 115.54 |
| **Cluster size 30** | 49.3273, 51.2727 | 67.6639, 46.177 | 514.657, 68.1797 |

Test Case 2 – More Misses than Hits

|  |  |  |  |
| --- | --- | --- | --- |
|  | **List Size 10** | **List Size 100** | **List Size 1000** |
| **Cluster size 1** | 7.85714, 8.52381 | 75.3731, 86.8458 | 750.375, 878.782 |
| **Cluster size 7** | 9.38095, 8.55556 | 76.5628, 58.3514 | 752.168, 558.572 |
| **Cluster size 30** | 24.9351, 26.0779 | 82.844, 65.8586 | 757.569, 526.641 |

\* Max value for test case 2 was (list size \* 2) + 1

**Conclusions**

For an in-order search that visits each element only once, searching with a self-adjusting list is always slower than linear search with a static container. This makes sense, because as the self-adjusting algorithm moves each accessed element to the front, all of the following elements are moved 1 step away from the head of the list. An in-order search never asks for an element more than once, so the self-adjusting function effectively just moves the next element further away from the head of the list, which is undesirable. This behavior is the same for a full list and a list with many misses.

For an in-order search that visits clusters of elements repeatedly, searching with a self-adjusting list is almost always faster than linear search with a static container. This makes sense, because this is exactly what the self-adjusting function is designed to do. It brings recently accessed elements to the front, which makes them much faster to find when they are searched for again. This behavior is the same for a full list and a list with many misses, though the benefit of a self-adjusting container over a static container diminishes as the number of misses increases. This is expected, because both searches have to visit every element in the list to verify that a value is not in the list.

An exception to the speedup of self-adjusting containers occurs when the size of the cluster (how many elements are searched for in each in-order group before moving on to the next group) is significantly larger than the size of the list. A very large cluster size creates a situation in which every element in the list is searched for in order, then one fewer elements are searched for in order, then one fewer, and so on. This is slow for the reasons discussed in the first paragraph of this section.

Generally, a self-adjusting list should only be used for searches if the values being searched for are frequently repeated in a short time-frame, and if the number of values being searched for is smaller than the size of the list. If these conditions are met, a self-adjusting container will speed up linear searches significantly. In perfect conditions, we measured a 10-fold speed increase when using a self-adjusting container.