Albert Wu

CptS 223

PA1 Report

1. Problem Statement

The problem was to read a list of integers and construct a sorted singly linked list containing the integers. At no point should the list be unsorted.

1. Algorithm design

To create the sorted list I repeatedly read the list of integers, inserting the minimum into the list each time, and used an unordered\_set to keep track of which integers I had already inserted. This took quadratic time as I had to run through the whole list for each integer.

As I had created a sorted list containing the integers, and I had access to both its head and its tail, finding the minimum and maximum was as simple as taking the head and tail respectively, which takes constant time. Slower methods to find the max or minimum would have involved keeping track of a running min/max and comparing it to each number in the list, which would have taken linear time in the number of integers.

In order to find the median, I calculated the length of the list subtracted by 1, then divided by 2 and rounded down, to get an index i. If the length of the list was odd, the median was the integer at that index. If the length was even, the median was the average of the integer at that index and the integer at the previous index (as there is no “center” item). In order to get the integers at the index I had wanted, I had to traverse the list to that index, which grows linearly with the size of the list. Hence, the algorithm was linear in the number of integers. A slower method to find the median would have involved alternately deleting ends from the linked list, which would have taken quadratic time as the nodes had no “previous” pointers.

Note: In order to avoid inserting duplicate items into the list, I used an unordered\_set to keep track of the indexes that had been inserted so far. Another option was to use read the whole list into an array first, and swap each chosen integer to the end, (then repeatedly read the array except the end) but I felt it was against the spirit of the problem and would still have taken quadratic time.

1. Experimental Setup

I repeated the experiment several times for input1.txt, and the timing statistics were all similar.

I tested on the WSU EECS servers.

Because I used unordered\_set and clock\_gettime(), I had to add the options -std=c++0x -lrt to the g++ command when running through terminal on the EECS servers.

The timer used has nanosecond precision.

1. Experimental Results and Discussion

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | time\_insert\* | time\_min\* | time\_max\* | time\_med\* |
| input1.txt | 655632 | 0 | 0 | 3 |
| input2.txt | 1259.69 | 0 | 0 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | min | max | med |
| input1.txt | 1 | 4000 | 2056.5 |
| input2.txt | 314 | 7999646 | 4001215 |

\*in milliseconds

For both input1 and input2, the timing results were similar. Finding the min and max both took under a nanosecond, which makes sense as they were both constant time operations. Finding the median took significantly less time then reading and inserting the integers, and the difference is much more noticeable in input2 (which has more integers). This also makes sense as while inserting is quadratic in the number of integers, finding the median is only linear.

The results also all make sense. All reported medians are between the reported min and max. The min and max reported for input1.txt seem to be correct based on a manual scan.