T-106.1227 Data Structures and Algorithms

Project Report (Final Part)

Cover Page

Course code: T-106.1227

Name: Nan Chen

Student number: 266035

Date: 11.05.2011

Table of Contents

1. INTRODUCTION

a) General description and aims (orientation, motivation, goals)

The project is to create and test the efficiency in terms of time and memory usage of a small C-language library of generic data structures modified from the source code provided by the book <Data Structures and Algorithm Analysis in C (Second Edition, Mark Allen Weiss)>[1]. This small library would not be a business-class comprehensive one, but for academic purposes, by creating it, going through and examining it, one should obtain detailed knowledge and hands-on experience of general data structures thoroughly. During the process of the project, one is supposed to form a systematic way of thinking in Data Structures and Algorithms with real implementation issues.

b) Justification of the library, exclusions if not feasible to evaluate all structures

Linked List, AVL Tree and several Sorting Algorithms have been evaluated. They were chosen as representatives of a branch of similar data structures.

Linked List is a common implementation of List in commercial languages, there is a built-in class – java.util.LinkedList along with JDK for Java, In C++'s STL library, List is implemented as a doubly-linked list. It is commonly chosen over Array-based List in the case that O(1) insertion / deletion running time is preferred.

AVL Tree:

Self-balancing binary search trees are of importance for search-intensive applications. Although JDK and STL chose red-black tree, it is still worthy to analyze AVL tree as well because of the similarities between the 2 data structures. AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

Sorting algorithms are generally significant for almost all information systems, for Java, we could find sort methods in java.util.Collections and java.util.Arrays.

2. LITERATURE SURVEY

a) General descriptions of the library

The library has been modified from the source code of [1]. Basically the modification arranges the source code in a single project by refactoring the structures, providing a possibility of reusing the source code as a data structure library. The correctness of algorithms is ensured by the existing testing code. The library code is located at: https://github.com/nanchen/c-data-structures-analysis/tree/master/ds-lib

b) List of data structure and algorithms

Evaluated data structures are highlighted

Data structure	Operations
List	
Linked List	MakeEmpty, IsEmpty, IsLast, Find, Delete, FindPrevious, Insert, DeleteList, Header, First, Advance, Retrieve
Linked List (cursor version)	Same as List
Stack (array version)	IsEmpty, IsFull, CreateStack, DisposeStack, MakeEmpty, Push, Top, Pop, TopAndPop
Stack (list version)	IsEmpty, CreateStack, DisposeStack, MakeEmpty, Push, Top, Pop
Queue	IsEmpty, IsFull, CreateQueue, DisposeQueue, MakeEmpty, Enqueue, Front, Dequeue, FrontAndDequeue
Tree	
Tree	MakeEmpty, Find, FindMin, FindMax, Insert, Delete, Retrieve
AVL Tree	MakeEmpty, Find, FindMin, FindMax, Insert, Retrieve
Top-down Splay Tree	MakeEmpty, Find, FindMin, FindMax, Initialize, Insert, Remove, Retrieve
Determinstic Skip List	MakeEmpty, Find, FindMin, FindMax, Initialize, Insert, Remove, Retrieve
Top-down Red Black Tree	MakeEmpty, Find, FindMin, FindMax, Initialize, Insert, Remove, Retrieve, PrintTree

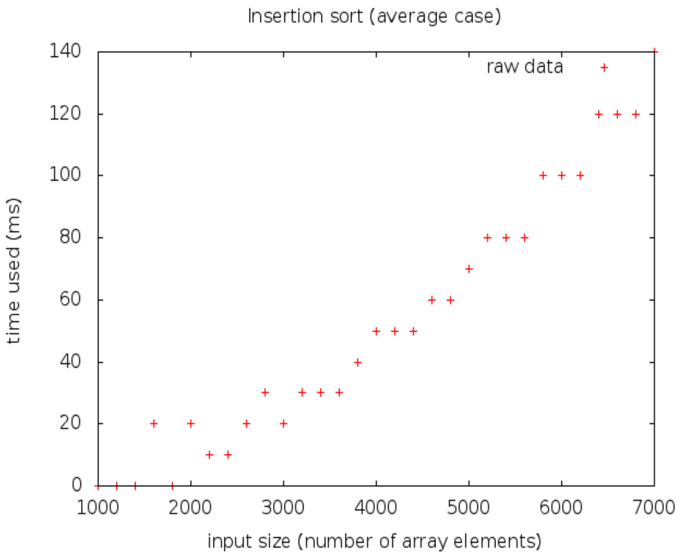
Treap	MakeEmpty, Find, FindMin, FindMax, Initialize, Insert, Remove, Retrieve
AA Tree	MakeEmpty, Find, FindMin, FindMax, Initialize, Insert, Remove, Retrieve
K-D Tree	RecursiveInsert, RecPrintRange
Hash Table	
Hash Table(separate chaining)	InitializeTable, DestroyTable, Find, Insert, Retrieve
Hash Table (quadratic probing)	InitializeTable, DestroyTable, Find, Insert, Retrieve, Rehash
Heap	
Binary Heap	Initialize, Destroy, MakeEmpty, Insert, DeleteMin, FindMin, IsEmpty, IsFull
Leftist Heap	Initialize, FindMin, IsEmpty, Merge, Insert, DeleteMin
Pairing Heap	Initialize, Destroy, MakeEmpty, Insert, DeleteMin, FindMin, DecreaseKey, IsEmpty, IsFull
Sorting	
Sorting Algorithms	InsertionSort, Shellsort, Heapsort, Mergesort, Quicksort
Other	
Disjoint Set	Initialize, SetUnion, Find

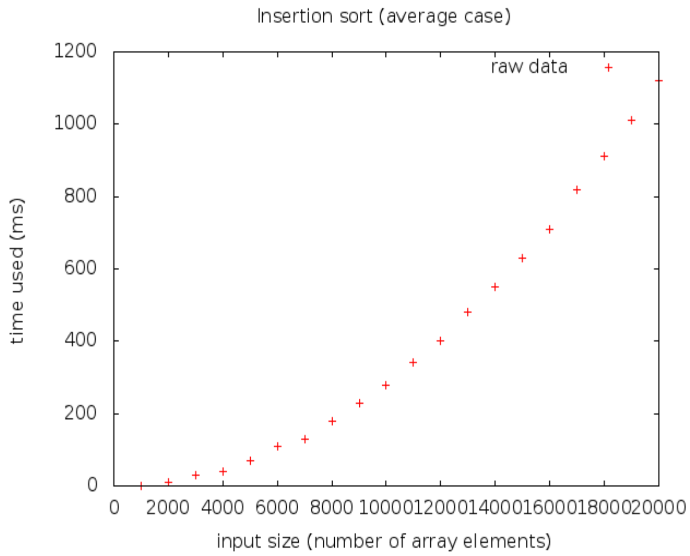
c) Working environment

ubuntu 10.10, gnu plot, gcc, eclipse, codeblocks, github.com, git scm

d) Evaluation methods (time and space)

Originally for time measurement it was planned to use “clock_t clock(void)” method provided by C standard library, but it had been found on Linux(Ubuntu 10.10 32-bit) the minimal unit is 10 ms. Thus the accuracy of this approach would be poor unless large input size per test is used. The left figure below illustrates the inaccuracy with small input sizes. In order to have acceptable accuracy, the input size must be larger, time used must be longer, as the right figure below shows:





Therefore, In this project, for recording time/space consumption, I mainly use a method of “embedding code” to achieve fine-grained measurement unit of time & space. On the other hand, the real time measurement (using “clock_t clock(void)”) is also supported by the measurement module.

A time/space measurement module (refer to https://github.com/nanchen/c-data-structures-analysis/blob/master/ds-lib/resourcetrack.c) is provided to log the time/space consumption, then in the source code of algorithms, the logTime() and logSpace() functions are written at appropriate places to record time/space usage.

For example in the following code snippet of insertion sort, corresponding stack space and time usage are logged, later on, will be used in analysis. Here I count all kinds of computations as “1 step” including addition, subtraction, multiplication, division, assignment, array index operations, pointer operations, entering / exiting function, etc.

Although those operations take quite different computation resources when computer really performs them, here for the purpose of algorithm analysis, this extra precision is not significant since for large enough inputs, the multiplicative constants are dominated by the effects of the input size itself [2].

As for space usage logging, not only the stack memory usage is recorded, the heap memory allocation/deallocation (malloc/free) is also recorded correspondingly.

```
void insertionSort(ElementType A[], int N) {
	//--------------space------------------------------
	const int SPACE = sizeof(ElementType) + sizeof(int);
	Resource_logSpace(SPACE); // log the stack space
	//--------------space------end---------------------
	int j, P; // time = 2
	ElementType Tmp; // time = 1
	for (P = 1; P < N; P++) { // time = 3
		Tmp = A[P]; // time = 2
		for (j = P; j > 0 && A[j - 1] > Tmp; j--) { // time = 7
			A[j] = A[j - 1]; // time = 4
			Resource_logTime(11); // 7 steps in for loop, 4 steps of retriving data from array & assign
		}
		A[j] = Tmp; // time = 2
		Resource_logTime(7); // log the time spent in this basic block (for loop)
	}
	Resource_logTime(5); // log the time outside of the for loop, including entering / exiting function cost
	Resource_logSpace(-SPACE); //log the deallocation of stack space when returning
}
```

After the time/space measurement code is embedded into the data structures' code, to determine the complexity class, I execute the operations with different input sizes (e.g. 10, 20, 30, … , 1000). After the execution of test, all the time/space usage for each different input size has been recorded, an analysis is executed to evaluate the time/space complexity of the operation. It performs the duty as the following:

1.) Calculate "constant" for common complexity classes (currently: $N^2; Nlog(N); N; \log(N)$ it could be easily extended to support other complexity classes) for each input size

    e.g for N^2, I calculate as: constant = $\frac{(time \text{ or } space)}{N^2}$ for each N

2.) Calculate the arithmetic mean (mean = $\frac{\sum constant}{N}$) of constant of each complexity class for all the input sizes

3.) Calculate the "Relative Standard Deviation" (RSD hereinafter) for each complexity class, choose the one with minimal RSD as the determined complexity class.

    RSD = $\sqrt{\frac{\sum (constant - mean)^2}{N}} \div mean$

4.) Draw the raw data and determined function on screen and compare them.

Refer to an approach with similar idea presented in sector 2.4.6 of [1]

e) Empirical results of the algorithms

Operation	Empirical time	Time RSD	Analytical time	Empirical space	Space RSD	Analytical space
			List			
List_makeEmpty	Time(N) = 4.10 * N	5.42%	O(N)	Space(N) = -7.96 * N	1.17%	O(N)
List_isEmpty	Time(N) = 4.00		O(1)	Space(N) = 4.00		O(1)

List_isLast	Time(N) = 4.00		O(1)	Space(N) = 8.00		O(1)
List_find-(worst-case) **[note]**	Time(N) = 2.05 * N	5.14%	O(N)	Space(N) = 8.00		O(1)
List_delete	Time(N) = 27.00		O(1)	Space(N) = -8.00		O(1)
List_findPrevious-(worst-case) **[note]**	Time(N) = 10.00		O(1)	Space(N) = 8.00		O(1)
List_insert-(onto-first)	Time(N) = 14.00		O(1)	Space(N) = 20.00		O(1)
List_deleteList	Time(N) = 4.05 * N	2.60%	O(N)	Space(N) = -8.00 * N	0.00%	O(N)
List_header	Time(N) = 2.00		O(1)	Space(N) = 4.00		O(1)
List_first	Time(N) = 3.00		O(1)	Space(N) = 4.00		O(1)
List_advance	Time(N) = 3.00		O(1)	Space(N) = 4.00		O(1)
List_retrieve	Time(N) = 3.00		O(1)	Space(N) = 4.00		O(1)
Sort						
insertion-sort-(average-case) **[note]**	Time(N) = 2.80 * N^2	6.61%	O(N^2)	Space(N) = 8.00		O(1)
insertion-sort-(best-case) **[note]**	Time(N) = 6.99 * N	0.33%	O(N)	Space(N) = 8.00		O(1)
insertion-sort-(worst-case) **[note]**	Time(N) = 5.51 * N^2	0.28%	O(N^2)	Space(N) = 8.00		O(1)
heap-sort-(average-case) **[note]**	Time(N) = 13.65 * NlogN	1.23%	O(N * logN)	Space(N) = 20.00		O(1)
merge-sort-(average-case) **[note]**	Time(N) = 23.38 * NlogN	2.32%	O(N * logN)	Space(N) = 4.71 * N	24.24%	O(N)
quick-sort-(average-case) **[note]**	Time(N) = 8.16 * NlogN	6.45%	O(N * logN)	Space(N) = 20.99 * logN	8.40%	O(logN)
shell-sort-(average-case) **[note]**	Time(N) = 13.63 * NlogN	10.91%	o(N^2)	Space(N) = 8.00		O(1)
AVL Tree						
insert	Time(N) = 49.35 * logN	13.17%	O(logN)	Space(N) = 12.28 * logN	7.77%	O(logN)
make-empty	Time(N) = 9.00 * N	0.05%	O(N)	Space(N) = -16.00 * N	0.00%	O(N)
find	Time(N) = 7.99 * logN	9.77%	O(logN)	Space(N) = 8.00		O(1)
find-min	Time(N) = 6.24 * logN	6.65%	O(logN)	Space(N) = 4.00		O(1)
find-max	Time(N) = 2.69 * logN	5.86%	O(logN)	Space(N) = 4.00		O(1)
retrieve	Time(N) = 3.00		O(1)	Space(N) = 4.00		O(1)

[Note]

Conflicting results are highlighted, see the comparison of shell sort below for reasons

For sorting algorithms, "average case" means sorting a "random" array of integers somewhat uniformly chosen at random between 2 integers (A formal proof to the uniformness is not given here). below is the C function "permute()" in use (provided by Mark A. Weiss with the source code of [1])

```
void Permute(ElementType A[], int N) {
        int i;
        for (i = 0; i < N; i++)
                A[i] = i;
        for (i = 1; i < N; i++)
                Swap(&A[i], &A[rand() % (i + 1)]);
}
```

"best case" for insertion-sort means doing insertion sort on a already-sorted array.

"worst case" for insertion-sort means insertion sort on a reversely-sorted (descending order) array; for List_find(), here it means finding the value of last element of the list; for List_findPrevious(), it means finding the previous element of last element of the list.

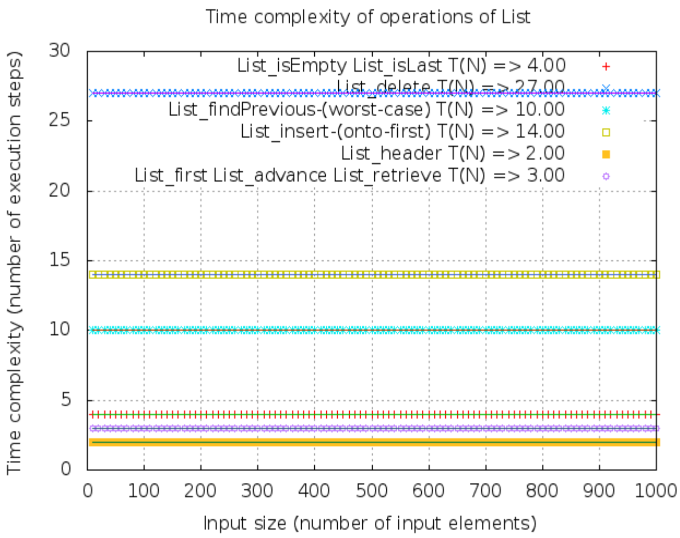
f) Comparison with the analytical results

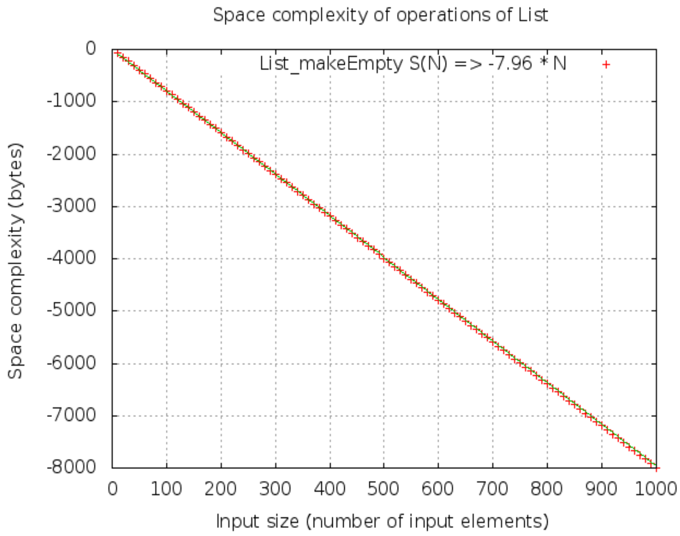
All the experimental points are plotted in diagram, then a "regression" function line/curve is plotted for comparison

List

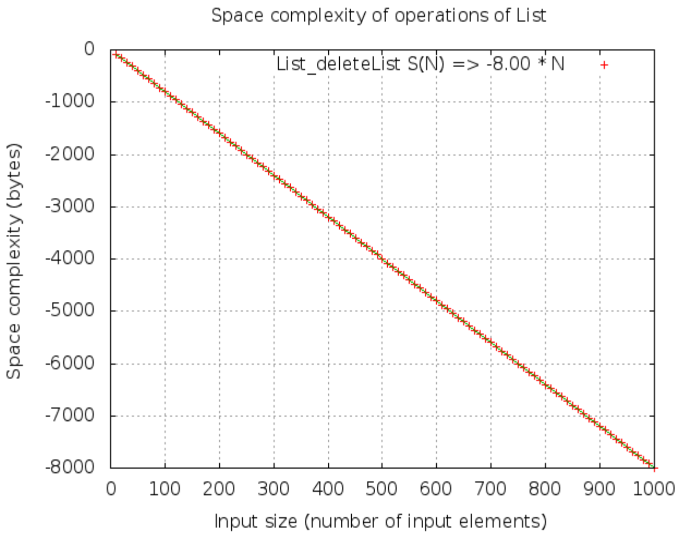
MakeEmpty, find (worst case: try to find the last element) & deleteList take time linear to the size of input in this experiment.

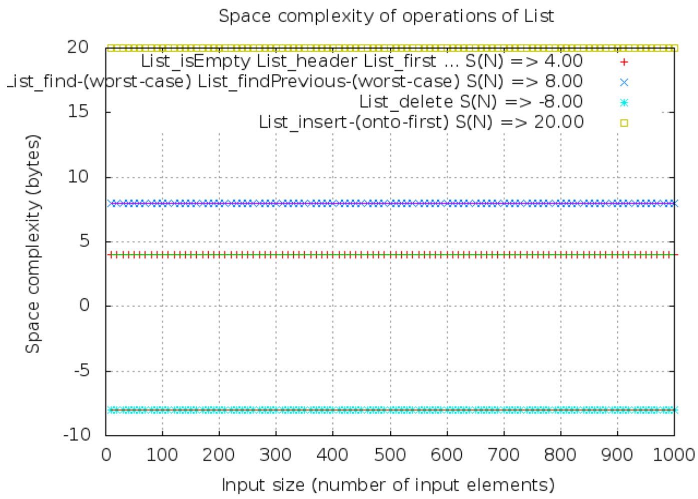
isEmpty, isLast, delete, findPrevious, insert, header, advance & retrieve all take constant time; makeEmpty frees space linear to the size of input





deleteList frees space linear to the input size; isEmpty, header, first, find, findPrevious, delete, insert all consume / free constant space

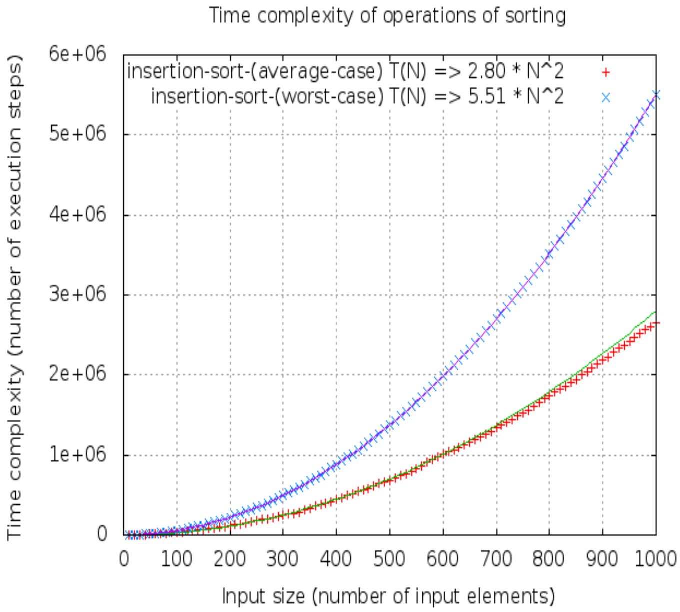


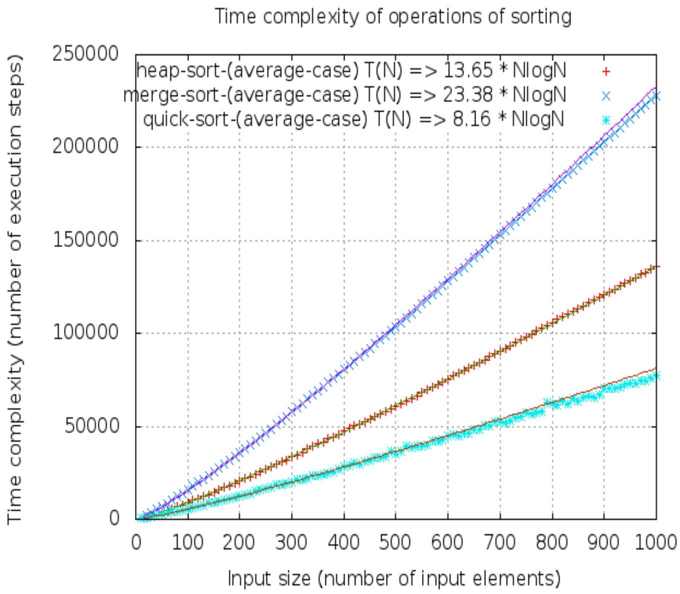


Sorting algorithms

Insertion sort in best case (input array is already sorted) takes time linear to input size in the experiment

Insertion sort in average / worst cases takes quadratic time; Heap sort, merge sort, quick sort take N*logN time





Conflicting result for Shell sort

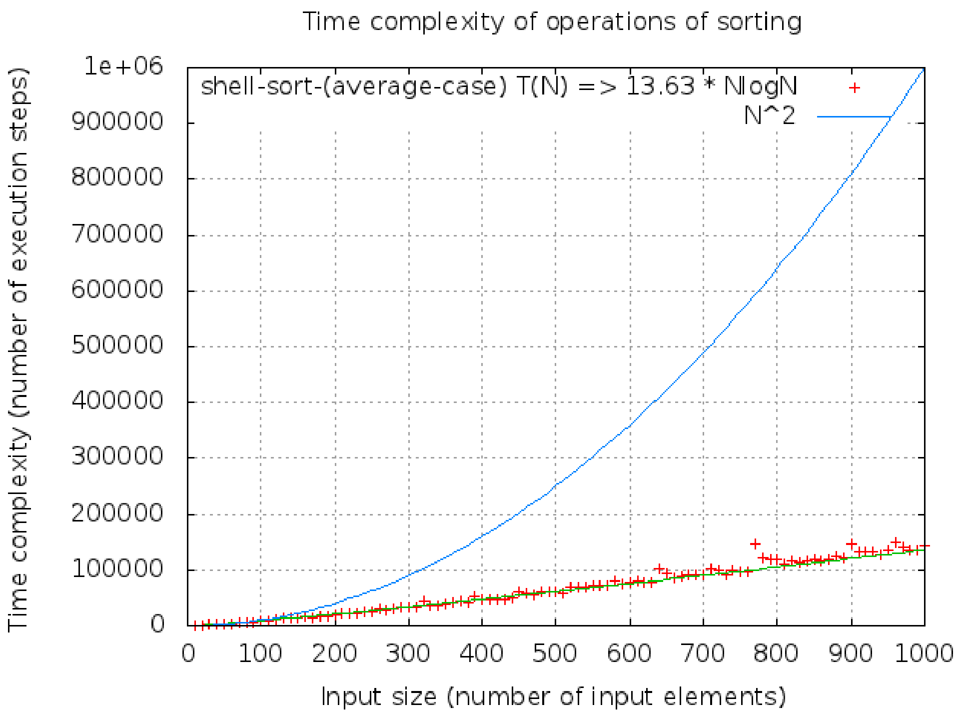
Although Shell sort is not a popular sorting algorithm, it is discussed in section 6.4 of [1] right after the discussion on insertion sort as a first try to break the quadratic time barrier of sorting algorithm. I analyzed it.

The analytical time complexity for Shell sort algorithm (using Shell's increments: h_t = floor(N/2), h_k = floor(h_{k-1}/2)) is o(N^2), ("There is an algorithm, Shellsort, that is very simple to code, runs in o(N^2), and is efficient in practice" -- chapter 6 of [1]) .

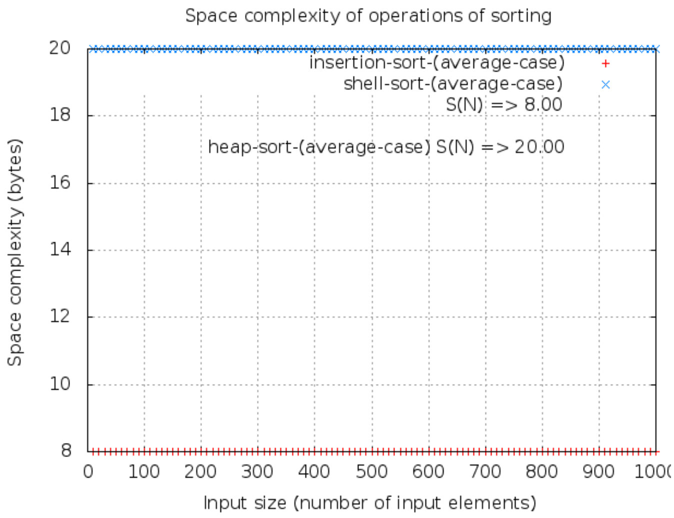
The analysis of Shell sort's running time is quite complicated. The running time of Shell sort depends on the choice of increment sequence. The average-case analysis is a long-standing open problem. Whereas an analysis and proof that the worst-case running time of shell-sort, using Shell's increments is Θ(N^2) is provided in 6.4.1 of [1]

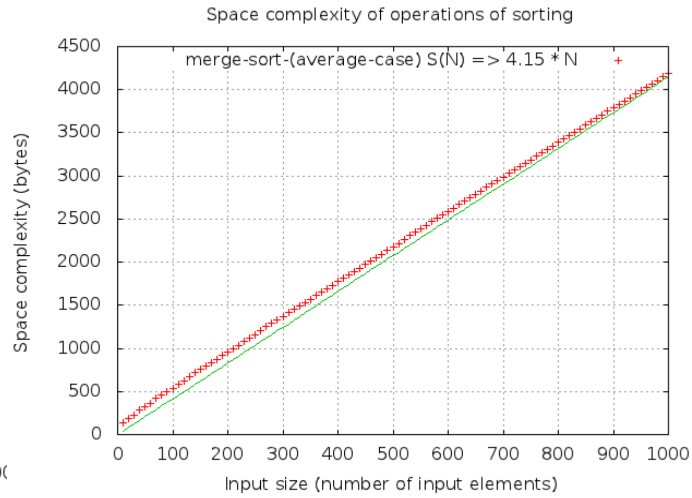
Here in the experiment, the determined complexity is NlogN, it is because most of the points are along with the 13.63 N*logN curve.

In the below graph, a curve of N^2 is drawn, it could be seen that T(N) = O(N^2) is correct. Furthermore, it is likely T(N) ≠ Θ(N^2), thus T(N) = o(N^2), conforming to the analytical result as the "o" notation in o(N^2) implies that the bound is not tight.

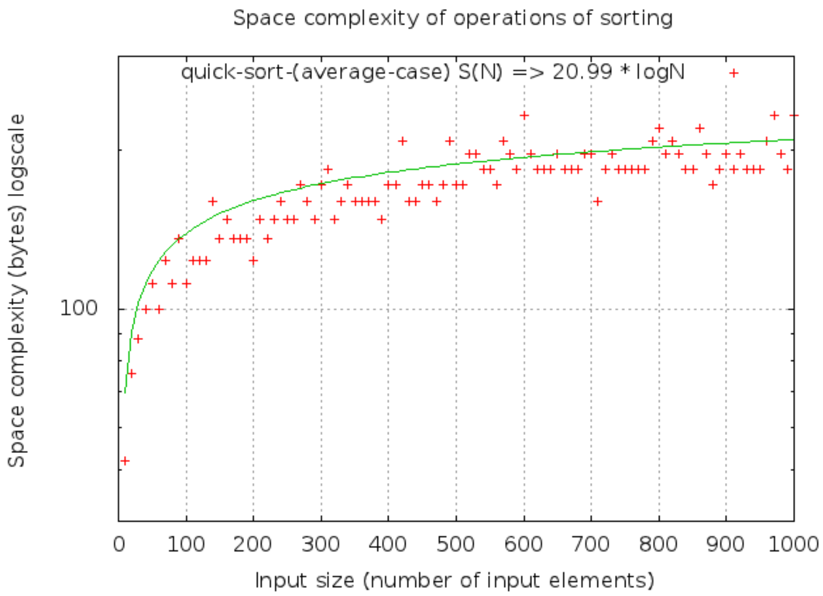


insertion sort (average case, worst case, best case), shell sort, heap sort all consume constant space; Merge sort consumes linear space.



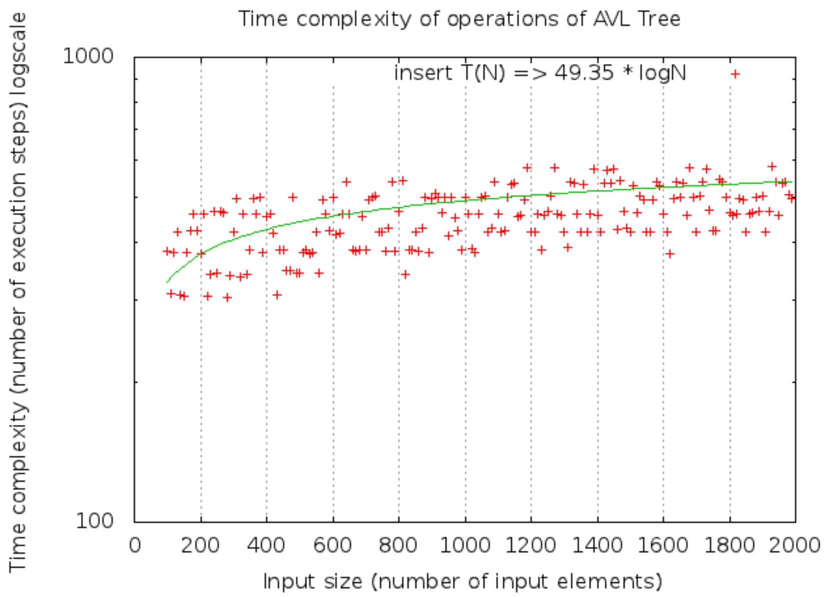


Quick sort consumes logarithmic space

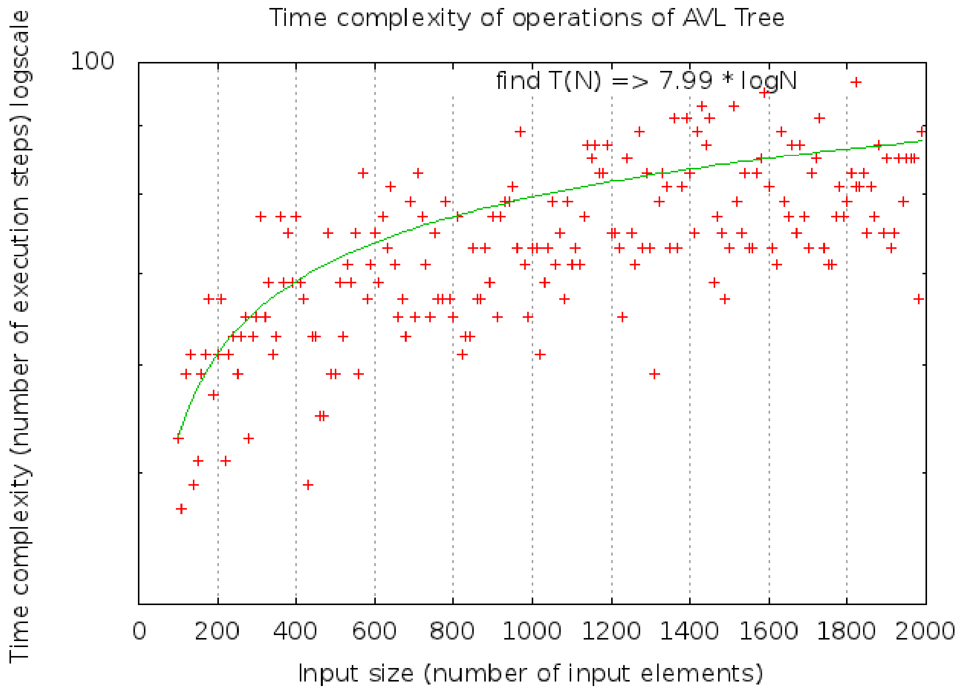


AVL Tree

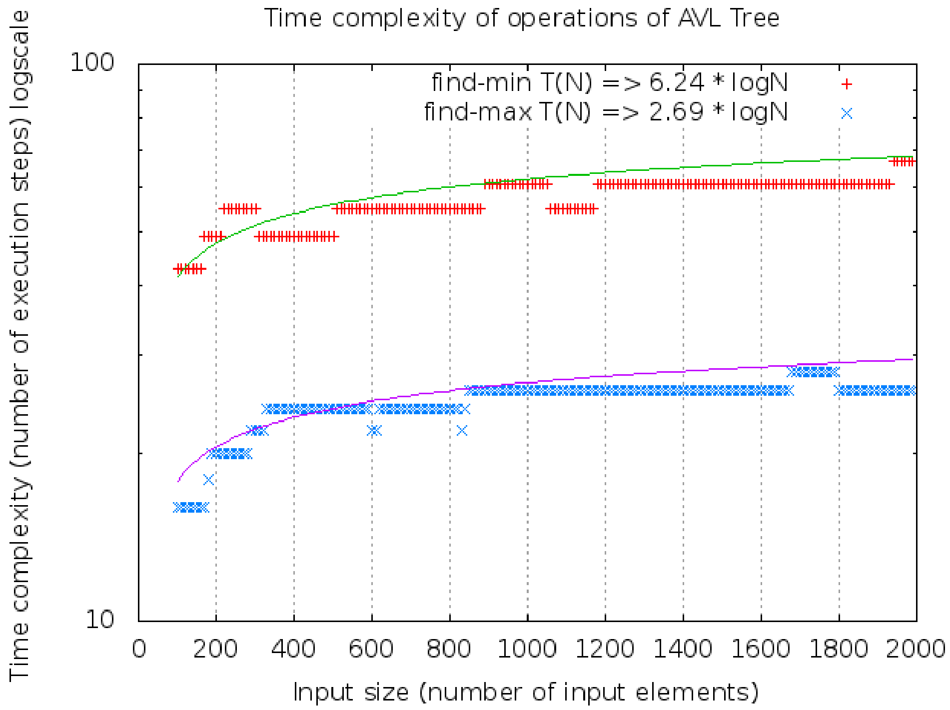
Insert takes logarithmic time, although the distribution of points is sparse(RSD = 13.17%), it could be seen the points are distributed alongside the regression logarithmic curve



find takes logarithmic time



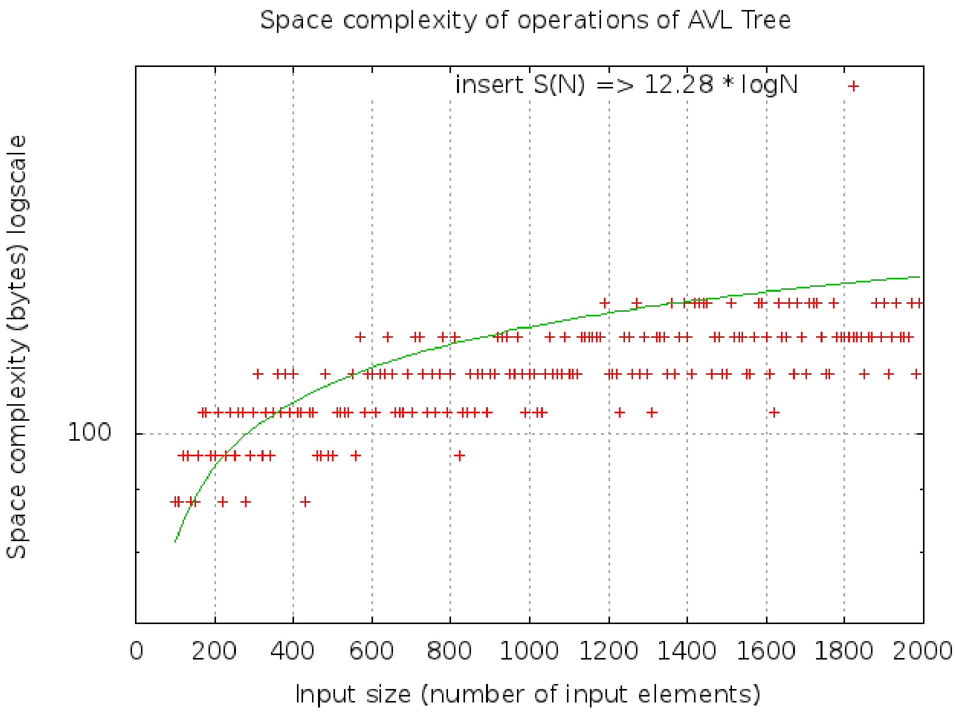
findMin and findMax take logarithmic time, there are some obvious “steps” on the diagram. It is because of some discrete depth differences of the min/max value with different input sizes when finding min / max



makeEmpty takes linear time; retrieve takes constant time

find, findMin, findMax all consume constant space; makeEmpty frees linear space

Insert consumes logarithmic space



3. DISCUSSION

a) Conclusions about the feasibility of the library, refer to the results above

Coverage:

The testing covers all the operations of chosen data structures / algorithms, including 12 cases for List, 6 cases for AVL Tree, and 7 cases for sorting algorithms.

List has been tested with input size of 10, 20, 30 … 1000, totally 100 different input sizes for each test case.

AVL Tree has been tested with input size of 100, 110, 120, … , 2000, in total 191 input sizes for each test case. Inputs are integers chosen uniformly at random between 0 and 10000.

Sorting algorithms have been tested with input size of 10, 20, 30, … , 1000, altogether 100 input sizes for each test case, inputs are integers chosen uniformly at random between 0 and 30000.

For some operations where inputs may affect the efficiency, “worst-case” inputs are applied, since the worst case execution gives an upper bound of resources the algorithm could consume, it is more important than “average case” and “best case”. For insertion sort, all these 3 cases have been evaluated. For other sorting algorithms, only “average cases” (randomly permuted integers) have been evaluated, since the “best case” and “worst case” differ for different sorting algorithms.

Thus, the coverage is considered to be sufficient.

Time/Space measurement approach:

In this project, for recording time/space consumption, the above-mentioned “embedding code” approach (in sector 2.d) is used to achieve fine-grained measurement unit of time & space.

As long as the code embedded into the evaluated data structures for recording time/space consumption has been carefully checked, it provides correct information about performance of data structures' operations.

Accuracy of automated evaluation mechanism:

The applied automated evaluation mechanism generally works fine, it currently could (based on “Relative Standard Deviation”) distinguish among 4 common complexity classes ($N^2; Nlog(N); N; \log(N)$), which is sufficient for all the test cases in this project, it could be easily extended to support more complexity classes.

Graph with regression line/curve:

At the final step of testing, points from raw experimental data and a regression line/curve according to the analyzed complexity class and arithmetic mean of the constant is drawn visually on a graph using gnuplot. This gives a illustrative way of showing the analysis of a test case.

In summary, the library provides functionality of basic data structures. It has same empirical complexity results with analytical ones on tested data structures and operations. Actually I am planning to publish an open-source data structure library based on it, targeting on embedded-system C development. Although it is still not a “business-class” library, I believe it has academic values.

b) Learning outcome

1.) Obtained better understanding of algorithm complexity, deeper knowledge of algorithm implementation

2.) Created a feasible automated method to analyze the complexity class of algorithms

4. TIME CONSUMPTION

Week	Duration	Hours	Phase	Content
			Defining Part	
9	28.02 – 04.03	9.71	Defining	Read the requirements, found a library to evaluate, investigated some measurement methods, wrote this report
			Solution Part	
10	07.03 – 13.03	12	Finding test method	Find solutions to test time/memory use, verify the solutions with one data structure
11	14.03 – 20.03	8	Creating library	Modify the source code, create the library containing basic data structures
12	21.03 – 27.03	21.5	Running test	Write all tests and run them
13	28.03 – 03.04	35	Analysis	Analyze the test results, modify test code if needed
14	04.04 – 10.04	21	Report	Write the report
			Final Version	
15 - 19	11.04 – 11.05	8.5	Final	Improve according to feedback
Total		**115.71**		

Work and time consumption log:

28.02 21:00 – 23:59 2 hours Read the assignment requirements
03.02 18:00 – 22:15 4.25 hours Evaluate libraries, investigate on measurement methods
04.03 17:32 – 19:45 2.21 hours Write defining part report
04.03 20:20 – 21:35 1.25 hours Finalize report and submit
09.03 8 hours Investigate on time/space measurement method
10.03 4 hours Determine to use “embedding code” method, analyze on real-time measurement drawbacks
15.03 8 hours Refactore existing source code into one project to create the library
22.03 8.5 hours Refactore library, develop time/space measurement module
23.03 5 hours Develop measurement module
24.03 8 hours Develop measurement module, call logTime & logSpace from list, test list.
29.03 7.5 hours Develop GNU plot script, improve measurement module
30.03 6 hours Test sorting algorithms, improve measurement module
31.03 7 hours Improve measurement module,
01.04 6.5 hours Test AVLTree, improve measurement module
02.04 8 hours Improve all the tests, analyse the results
04.04 6.5 hours Improve embedded logging code
05.04 6.5 hours Write report, improve code
09.04 8 hours Write report
10.05 4 hours Improve report according to comments
11.05 4.5 hours Finalize report

5. REFERENCES

Homepage on github: https://github.com/nanchen/c-data-structures-analysis

Experimental data: https://github.com/nanchen/c-data-structures-analysis/tree/master/plot

[1] Mark A. Weiss. Data Structures and Algorithm Analysis in C (Second Edition) Published by Addison-Wesley, 1997 ISBN: 0-201-49840-5

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, Second Edition. Published by MIT press, September 2001, ISBN-10: 0-262-53196-8