POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Report

DYNAMIC DATA STRUCTURES

Adam Korba, 151962

Teacher
PhD. Grzegorz Pawlak

POZNAŃ 2022

Contents

Chapter 1

Introduction

Goal of this report is to compare three dynamic data structures:

- Sorted Linked List (LL)
- Binary Search Tree (BST)
- Balanced Binary Search Tree (AVL)

Our task was to implement them in a programming language, measure and compare their performances in the following operations:

- Inserting elements to the structure
- Removing elements from the structure
- Searching for elements

Chapter 2

Exercises

2.1 Exercise 1 - Implement structures

2.1.1 Implementation details

My programming language of choice was C++. I mostly used a standard library and some of the features from STL (Standard Template Library) like `<vector>` header and `<algorithm>`. Structures are implemented with pointers to maximize efficiency. For testsing I used unit test framework called "googletest".

2.1.2 Interface

My first step was to implement interface class in C++ called TestSubject, my other it has defined following three virtual methods:

- virtual void insert(Person p) = 0;
- virtual Person search(int index) = 0;
- virtual void remove(int index) = 0;

As well as some utility methods that will be used later to measure the times of performing these operations.

2.1.3 Linked List

Linked list is a very simple structure where every node stores pointer to next one. In sorted linked list time complexity of most operations is linearly dependent on number of elements in the structure. The reason in worst case it is required to iterate over all elements in list to perform operation.

Table 2.1: Linked list time and space complexity

parameter	complexity
insert	$\mathcal{O}(n)$
search	$\mathcal{O}(n)$
remove	$\mathcal{O}(n)$
additional space	$\mathcal{O}(n)$

2.1.4 Binary Search Tree

the binary search tree is a more complex structure than Linked List every node stores pointers to two child nodes. Nodes are structured in a way that in the "left" subtree all elements have smaller value of data than root and in the "right" subtree all values are higher. This way of organizing data allows for really fast search of elements, because we traverse tree in "divide and conquer" fashion.

Table 2.2: BST expected time and space complexity

parameter	complexity
insert	$\mathcal{O}(\log n)$
search	$\mathcal{O}(\log n)$
remove	$\mathcal{O}(\log n)$
additional space	$\mathcal{O}(n)$

It is worth to mention that we have three basic ways of tree travesal:

- Pre-order - recursive function returns elements in this order: (Left, Root, Right) - used often to copy tree
- In-order - (Root, Left, Right) - gives a sorted sequence
- Post-order - (Left, Right, Root) - used often to delete tree

2.1.5 Balanced Binary Search Tree

This is an improved version of BST. These two types are most common:

- BBST - number of elements of the two children subtrees of any node differ by at most one
- AVL Tree - the heights of the two children subtrees of any node differ by at most one

I decided to implement AVL tree which required changing an insert method a little in a way that a tree balances itself after adding any element (using rotate left and rotate right functions). Other than that this structure inherits most of a functionality from regular BST. Complexity should be also logarithmic but I expect it to be a little worse in insert time (as we need to balance a tree) and a little better in search and remove time.

2.2 Exercise 2 - Prepare data sets

2.2.1 Data generation

In my C++ program class Person has following attributes

- index (7 digit index number)
- first_name (`std::string` that stores firstname)
- last_name (`std::string` that stores lastname)

To create data for experiments python script was used. Module "names" served to generate random firsname lastname pair, then they are given next index number, so the following data is sorted, it gets scrambled later in C++ with `std::random_shuffle()`

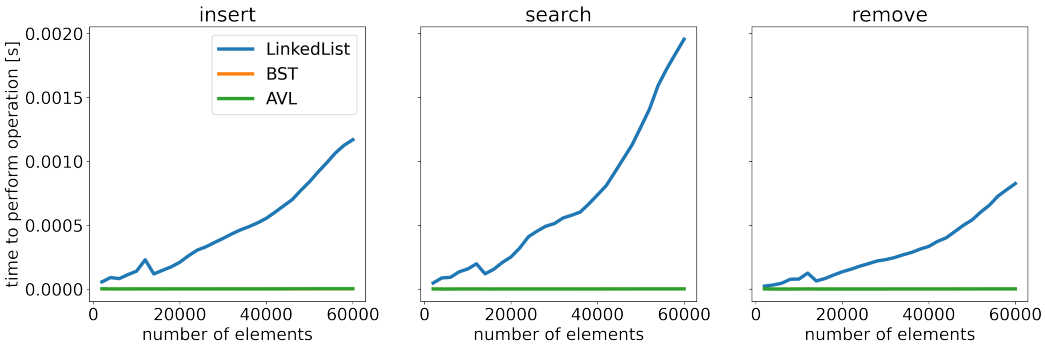

FIGURE 2.1: Results of experiment 1

2.3 Exercise 3 - Compare implemented structures

2.3.1 Remarks

Time was measured using `clock()` function from C header `<time.h>`. Every data point was computed by finding an average time needed to perform operations on all n elements.

Linked list happened to be much slower than binary search trees, that is why I decided to present results in two Figures. Fig 2.1 shows all three structures for number of elements ranging between 0 and 60000, while Fig 2.2 shows only binary search trees for input sizes ranging to one million. At the end of this PDF in Figures 3.1 and 3.2 you can find some selected data points in tables.

2.3.2 Linked list

All operations required linear time to perform. We can see that search performed worse than insert and remove but the reason for this phenomenon is that we take the average times of adding items to a list increasing in size from 0. While in search all measurements were taken for the n-sized list. In remove, we have a reverse situation.

2.3.3 Binary search trees

Both trees performed very well even for big sizes of inputs (with logarithmic time). Insert in AVL is worse than in regular BST, because of a need to rebalance a tree after insertion. But AVL performed better in searching and removing elements.

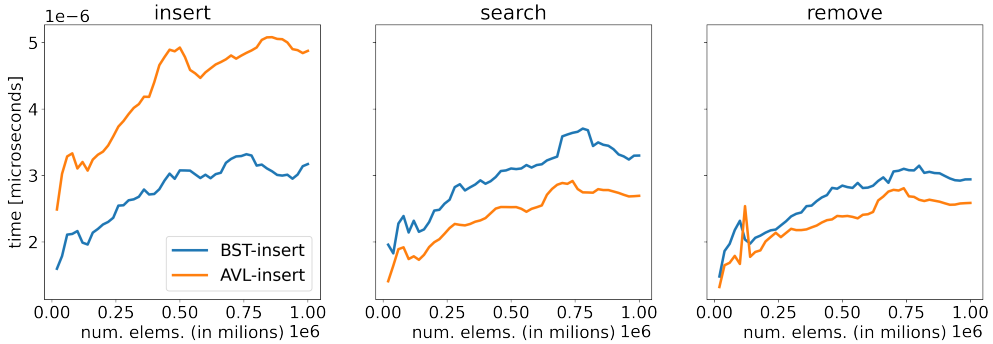

FIGURE 2.2: Results of just BST and AVL

Chapter 3

Conclusions

Experiments have shown that the performance of sorted linked lists is much worse than binary trees. The only advantage is that implementation is much easier, but if possible one should always choose BST over it. When it comes to choosing AVL over regular BST one should take into account some factors. If inserts are performed very often then probably regular BST would be a better choice, but if searching is the main priority then one could benefit greatly from using an automatically balancing tree like AVL.

Table 3.1: Some datapoints from experiment 1.

structure	AVL	AVL	BST	BST	LinkedList	LinkedList
operation	insert	search	insert	search	insert	search
n						
2000	2.9145e-06	1.649e-06	1.9525e-06	1.93e-06	5.67785e-05	4.8002e-05
10000	2.2218e-06	1.3668e-06	1.3999e-06	1.4576e-06	0.000140588	0.000157958
20000	2.55645e-06	1.3485e-06	1.5669e-06	1.6297e-06	0.000415381	0.000548462
30000	2.71697e-06	1.60143e-06	1.78383e-06	1.9281e-06	0.000508767	0.000613896
40000	3.131e-06	2.13363e-06	1.824e-06	1.84533e-06	0.000729945	0.000966246

Table 3.2: Some datapoints from experiment 2.

structure	AVL	AVL	BST	BST
operation	insert	search	insert	search
n				
40000	3.02355e-06	1.64045e-06	1.78515e-06	1.82688e-06
100000	3.104e-06	1.74117e-06	2.16187e-06	2.1407e-06
200000	3.88445e-06	2.19047e-06	2.34565e-06	2.46095e-06
500000	4.4415e-06	2.43856e-06	2.74066e-06	2.94459e-06
1000000	4.93734e-06	2.70508e-06	3.0813e-06	3.34094e-06

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.