Exam Project – Part 2

DM803 Advanced Data Structures

Dennis Andersen – deand17

Department of Mathematics and Computer Science
University of Southern Denmark

May 8, 2022

1 Introduction

The goal of the second part of the exam project was to implement a randomized search tree and a partially persistent list, and investigate their properties, specifically examining the average search complexity and variation in search complexity of the randomized search tree and for the partially persistent list, the connection between the number of extra pointers allowed in a node when using the node copying method with the amount of space consumed by the data structure. We start by briefly discussing design choices for the implementation and then proceed to discuss experimental investigations.

2 Implementation

The randomized search tree and partially persistent list data structures have both been implemented using C++. The randomized search tree supports the operations described in the project description, where

`search` returns a pair where first is a pointer to the node with the given search key if it exists and `nullptr` otherwise, and second is the depth of the node the key resides in.

`insert` returns a pair where if the key was not already in the tree then first is a pointer to the newly inserted node, and second is true, otherwise, first is `nullptr` and second is false.

`split` splits the original tree in two, modifying the original tree to contain all the keys smaller than the given key k and returning a new tree t with all keys larger than k.

`merge` returns a new tree that is the merge of the two given trees. The two trees given as arguments are both emptied in the process, leaving them in a state equivalent to calling the clear() method.

`erase` returns a pair where if the node x with search key k was found, the first is a pointer to the new root that is the merge of x's two children and second is true. Otherwise, first is `nullptr` and second is false.

The partially persistent list supports the operations described in the project description, where

`newversion` simply increases the version number by one and adds two additional pointers to the two access lists kept by the list (one for primary nodes and one for associated nodes).

`search` returns a pair where first and second is the key and associated key at the ith position respectively if they exist. If there is no ith index, then each have the special value `none`, which is simply an `enum` defined to have the smallest 32-bit integer value, that is $-(2^{32}-1)$.

`insert` inserts the key k as the new ith element in the list, i.e., between the (i - 1)st and ith element of the current version and returns true if the key was inserted and false otherwise.

`update` updates the key in the ith element to the given key in the newest version, and also the key in the associated item, if present. If an associated key is not present and an associated key k' is given, an associated node is created. Returns true if the update succeeds and false otherwise.

Also provided with the partially persistent list is a `show_info` method, which simply prints a summary of the current state of the list, i.e. version, size, number of nodes, associated nodes, extra pointers and a summary of the memory consumption. Also, each data structure support basic contain methods such as `size`, `empty`, and `clear`, as well as a method to "stringify" the data structure, allowing the contents to be "pretty printed" to the terminal. However, this has the side effect that printing is only suitable for objects containing a handful of elements (say 10 for example).

The test programs for the randomized search tree and the partially persistent list are `rtree_test` and `pplist_test` respectively. Each support an interactive mode where the above operations can be tested and the data structures printed to the terminal to examine their contents. To see a list of possible commands, simply enter `H` or `h` after running the program.

The randomized search tree implementation is based on the pseudo code for binary trees and pseudo code for rotations from chapters 12 and 13 respectively of CLRS, as well as the descriptions provided in the project description. Priorities are generated as uniformly random unsigned long long integers (meaning 64 bit unsigned integers), with 0 being reserved as a dummy priority to insert when splitting the tree.

The partially persistent list implementation is essentially also a one-to-one implementation of a singly linked list augmented by the node copying method described in the paper. The main implementation detail are that we use an unordered map to hold extra pointers, which gives the same constant access time as an array or list, since we only store a small number of extra pointers (16 to be precise) and using a method from the Boost library the chance of collisions is minuscule.

A makefile is included and the default target builds the programs for both the randomized search tree and the partially persistent list as `rtree_test` and `pplist_test` respectively. The implementation has been tested to work on the computer lab machine `imada-106333`. In addition, there are python scripts for each data structure to generate input test files, with each script generating two uniformly random samples of the integers $0, \ldots, n-1$ are generated, the first of which is used for n insert operations followed by n search operations using the second sample. Also included is a small shell script `test.sh`, taking the following options:

- `-h` Print this Help.
- `-n` Number of keys to insert. Defaults to 1024.
- `-k` Number of times to run search for n keys. Defaults to 1.

This script runs the `rtree_test` and `pplist_test` programs on each of the k input files, followed by a post processing step, which outputs the results of the test to `stdout` unless redirected.

3 Experimental Results

We now want to investigate and understand the randomized search tree and partially persistent list data structures. We first examine the average search complexity and then the variation in search complexity of our implementation of the randomized search tree, where we measure complexity as the depth of the node a key resides in. We then examine space complexity as a function of the maximally allowed number of extra pointers p in a node for our implementation of the partially persistent list, where we measure space complexity as the number of nodes and total amount of

memory used. To run our tests we first used the respective python scripts to generate input files, such that the same inputs were used for different tests and then used the script `test.sh` described in the previous section for the actual testing and post-processing.

3.1 Average Search Complexity

For testing the average search complexity, we have used $k = 3$ for all our tests, meaning for each input size $n \in \{2^{10}, 2^{11}, \ldots, 2^{15}\}$, we have made 3 test runs and then calculated the average depth of the node containing the search key across the 3 runs.

Figure 3.1 shows the complexity as a function of n for the search operation on the randomized search tree, with the plot using a lg-scale on the x-axis. The best case average complexity for the search operation is $\lg n$, thus we would like our data points to form a line, which is more or less also what we observe. Using logarithmic regression in R, we can estimate the constant term in front of the logarithm as 1.343 with a p-value of 0.0005, indicating a high confidence in a logarithmic average search complexity.

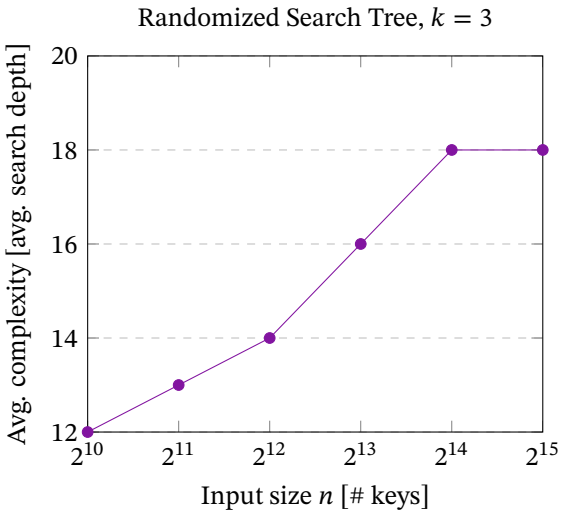

Figure 3.1 Average search complexity (search depth) for our implementation of the search operation for the randomized search tree. The plot uses a lg-scale x-axis and shows the average search complexity across $k = 3$ test runs for increasing input sizes n. Shows logarithmic behaviour for all tested input sizes.

3.2 Variation in Search Complexity

We now look at the variation in search complexity for the randomized search tree. That is, for n searches, we make a histogram counting the number of searches s that used search depth d and then accumulate the percentage of searches that had complexity d as shown in the plot in figure 3.2. For this test we used $n \in \{2^{10}, 2^{11}, \ldots, 2^{15}\}$ and $k = 3$ test runs and used the average search depth across the 3 runs.

The variation in average search complexity for increasing input sizes is shown in figure 3.2. For all input sizes we observe only a small variation, with 54.72% using a search depth of 11 or less for $n = 2^{10}$ out of a maximum search depth of 23. For $n = 2^{15}$, 56% of searches reach a search depth of 18 or less out of a maximum search depth of 37.

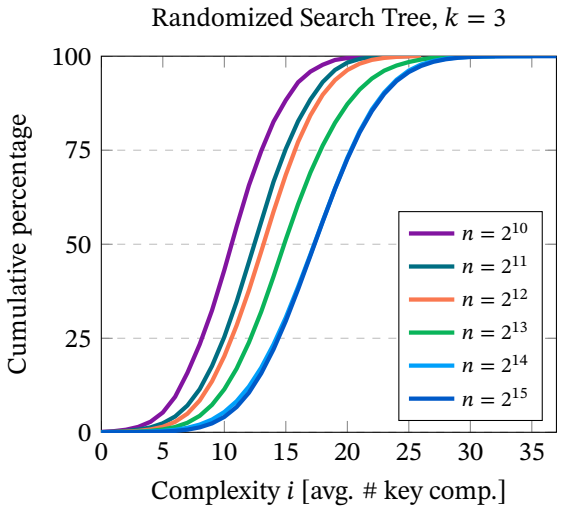

Figure 3.2 Variation in average search complexity (search depth) for our implementation of the search operation for the randomized search tree. Shows the variation in average search complexity across $k = 3$ test runs as the cumulative percentage of searches that used i comparisons or less. Only a small variation in the average search complexity is observed across the tested input sizes.

3.3 Extra Pointers and Space Used

Our final point of investigation is on the connection between the number of extra pointers allowed in a node and the amount of space used as a function thereof. As with the previous tests, we have used the average of $k = 3$ test runs. We investigated this connection experimentally, by first allowing only a single extra pointer in each node and then doubling the allowed number in each subsequent

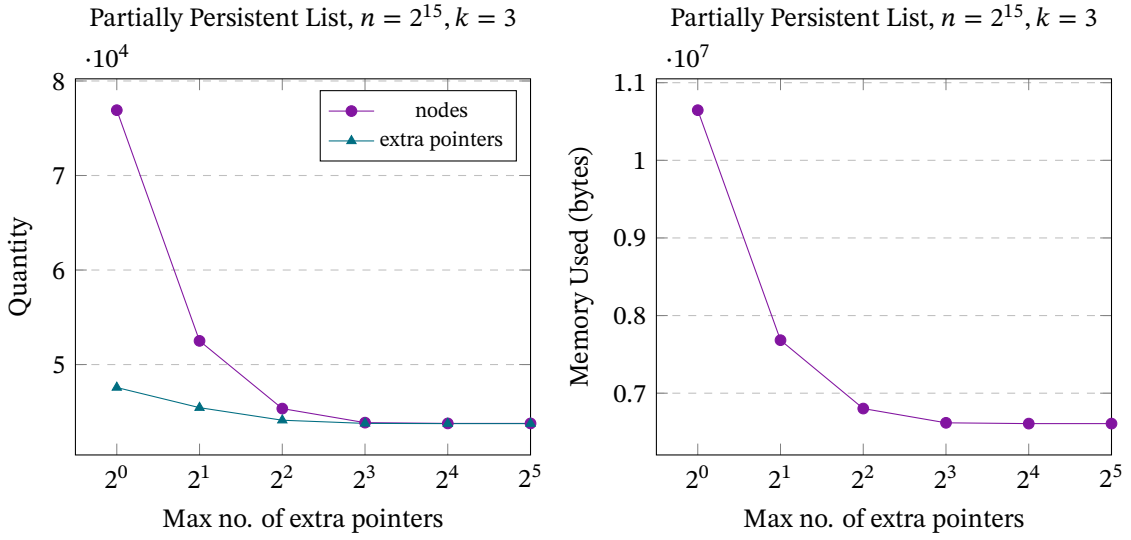

Figure 3.3 Space complexity as a function of the maximally allowed number of extra pointers p in a node, where $p \in \{1, 2, \ldots, 32\}$. Left shows the number of additional nodes and extra pointers used. Increasing the allowed number of extra pointers from 1 to 4, shows a large reduction in the number of nodes created by 41%. However, going from 4 to 8 only shows a 3% further reduction, and from 8 to 16 only a 0.03% reduction. There is no improvement from 16 to 32 and 16 extra pointers showed the minimum number of nodes and extra pointers used. Right shows memory consumption in bytes. A 36.1% reduction is seen by increasing the number of allowed extra pointers from 1 to 4. Going from 4 to 8 only shows a 3% further reduction and from 8 to 16 a marginal 0.002% further reduction. There is no improvement from 16 to 32.

test until we saw no improvement. For each test, we measured the number of extra pointers used, the total number of nodes created and the total memory consumption in bytes. When generating the input test files, we used a probability of 0.3333 that an associated key was generated.

Figure 3.3 shows the space complexity as a function of the maximally allowed number of extra pointers p in a node, where $p \in \{1, 2, \dots, 32\}$. Figure 3.3 left shows the number of additional nodes and extra pointers used. Increasing the allowed number of extra pointers from 1 to 4, shows a large reduction in the number of nodes created by 41%. However, further increasing the allowed number of extra pointers from 4 to 8 only shows a only 3% further reduction, and from 8 to 16 only a 0.03% reduction. However, the minimum number of nodes and extra pointers used is observed when allowing 16 extra pointers. There is no improvement going from 16 to 32. Figure 3.3 right shows total memory consumption in bytes. A 36.1% reduction is seen when increasing the number of allowed extra pointers from 1 to 4. Further increasing the allowed number of extra pointers from 4 to 8 only shows a 3% further reduction and from 8 to 16 only a marginal 0.002% further reduction. However, the minimum amount memory consumed is observed when allowing 16 extra pointers, and again there is no improvement going from 16 to 32 extra pointers.

Note that in figure 3.3, we have only included the results for the largest input size tested, that is, $n = 2^{15}$, although we also tested with smaller input sizes, since the results were proportionally comparable. For example, with only 1 extra pointer allowed, for $n = 2^{15}$, 76,911 nodes were generated, which is corresponds to $2.3471n$, and for $n = 2^{14}$, 38,368 nodes were generated, which is corresponds to $2.3418n$, and so on. In comparison, with 16 extra pointers allowed, for $n = 2^{15}$, only 43,772 nodes were generated, corresponding to $1.3358n$, which strongly correlates with the probability of 0.3333 of an associated node.