Mahidol University
International College

ICCS315 Applied Algorithms

Assignment 1 Report

Written by
Krittin Nisunarat 6280782

30 Jan 2022

Contents

Chapter 1

Resizeable Arrays

Traditionally, resizable array has a high amortized cost depending on α for expansion and shrinking. HAT array theoretically fares with a constant amortized cost on expansion and shrinking. Our goal is to perform teh benchmarking of the actual implementation of traditionaly resizable array and Sitarski's HAT array.

1.1 Set up

Both of data structure implementations are written in C++ which runs on ubuntu server. The actual implement is in Github.

1.2 Append latency

In this section, the cost of appending one key to the array mainly comes from reading memory. On tradditional resizable array, pushing new key to the back costs approximatetly 47 cycles from reading memory pointer which needs to be conducted at most the entire array. On the other hand, Sitarski's HAT array takes less cycle per apeend operation which is 26 cycles since the entire array is separated into b size where b where $b = 2^k$.

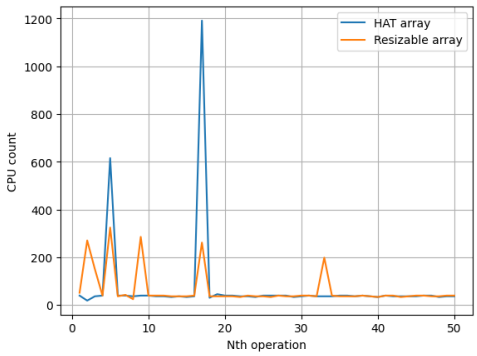


Figure 1.1: Benchmark of each append costs

Figure 1.1 shows the cost of each append out of 50 append operations. Tradditional

resizable array shows up to stike CPU cycle more on each operation since the expansion happens more often than HAT array. However, HAT array costs significantly higher cycle because the resizing needs to be done in both levels combine with copying data over.

In conclusion, traditional resizable array takes more cycle to access memory and append new day than HAT array. On the expansion, tradditional resizable array is better than HAT array.

1.3 Access latency

Getting element by index in HAT array is tricky in the way that we need to calculate the block index and the index of element in particular block. This seems to take more cycle than traditional resizable array. Surprisingly, HAT array uses slightly less cycle then resizable array. Benchmark is finding the average (out of 100,000 times) CPU count on each data structure element access. The result is that HAT array takes 33 cycles on average while traditional resizable array takes 38 cycles. Therefore, HAT array access element faster by 8 percents approximately.

1.4 Scan throughput

Continue from section 1.2, this section will illustrate the total latency of accesing/reading all elements in array. Let's start with both arrays have 500 elements. HAT array costs 16800 cycles in total while traditional resizable array costs 18918 cycles on reading.

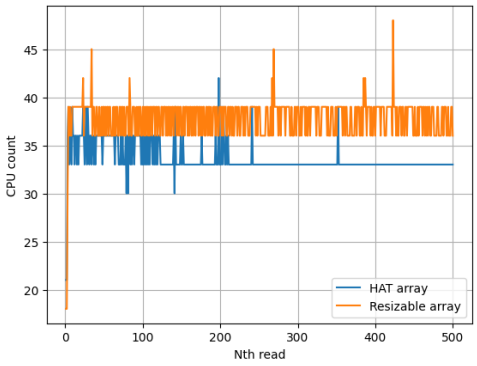


Figure 1.2: Benchmark of scanning through array

As we can see in figure 1.2, HAT array is nearly stable on reading after 300 elements. Resizeable array, on the other side, have a few operation that strike CPU cycle which reaches over 45 cycles. In conclusion, HAT array still wins in this competition that it scans faster by 13 percents.

1.5 Overall throughput

In section 1.1, I have shown append latency which conclude to have different benefits. For this benchmark, I will test inserting 50 elements to both tradditional resizable array

and HAT array. As a consequence, resizable array takes 1153 cycles on average while HAT array takes 1114 cycles on average. Even though the expansion is costly in HAT array, overall throughput of it is around 5 percents faster than resizable array.

Chapter 2

Skip lists

2.1 One directional skip list vs. sorted linked list

Sorted linked list theoretically takes $O(n)$ complexity for insertion and scan throughput. Skip list comes to take place in theory land that it improve time complexity of those 2 operations. For insertion, skip list takes $O(\log_2 n)$ with whp. The same thing goes with scan throughput since we guarantee with high probability that every access costs $\Theta(log_2 n)$ where n is the number of elements. Let's see the real breanchmark using c++ language on insertion and scan throughput.

2.1.1 Append latency

Chapter 3

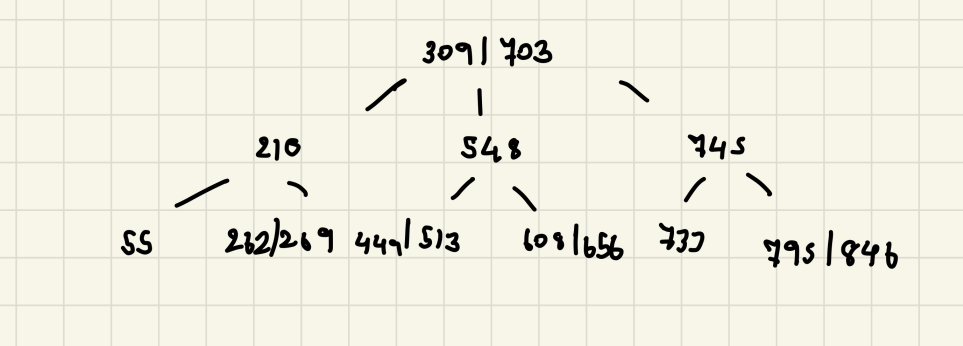
(a,b) tree

3.1 Multiple keys insertion.

Starting with an empty tree, we want to insert the following keys:

733, 703, 608, 846, 309, 269, 55, 745, 548, 449, 513, 210, 795, 656, 262

The result of *(2,3) tree* is



3.2 Key deletion

Suppose that we want to delete 309 in *(2,3) tree* (Firgure 3.1), it falls into case 2 which we need to merge from sibling (Figure 3.2).

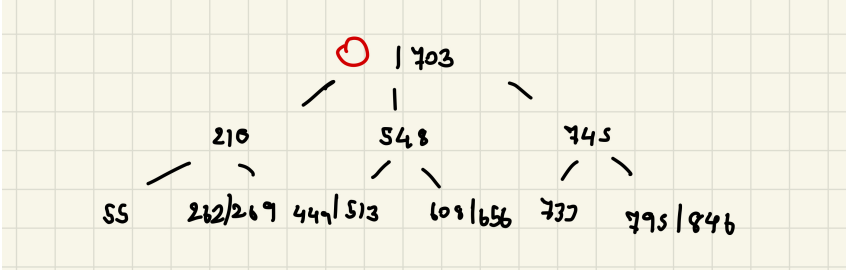


Figure 3.1: Unbalanced tree without key 309

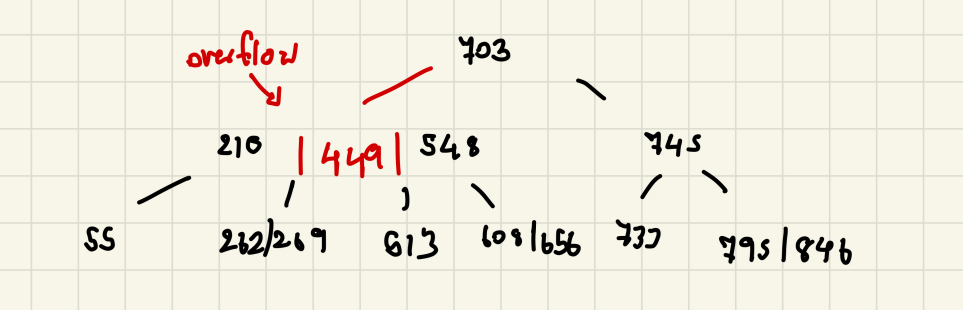


Figure 3.2: Tree after merging

However, the merged tree in figure 3.2 is not balance as the second layer node has 3 keys. So, we need to project 449 up to root.

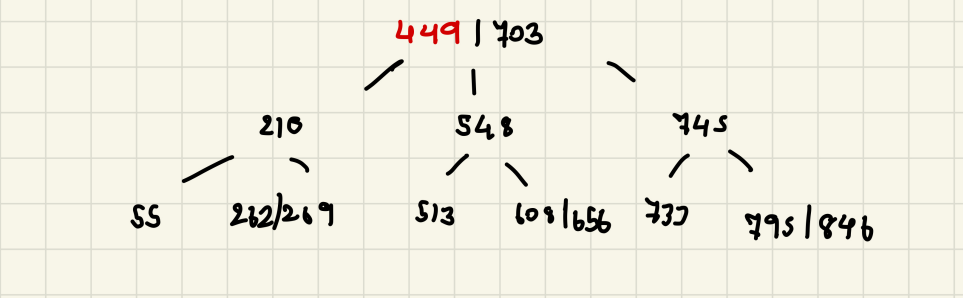


Figure 3.3: Final tree

As you can see in Figure 3.3, the replacements are 449 which is the result from merging non-root tree in figure 3.2.

$$\alpha_{\text{me}} = a - 1$$
$$\alpha_{\text{sib}} = a$$
$$\alpha_{\text{sib}} + \alpha_{\text{me}} = 2a - 1 \leq b$$

Chapter 4

B-tree speed