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 $b = 2^k$.

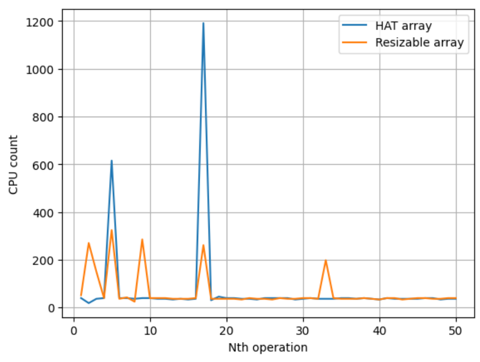


Figure 1.1: Benchmark of each append costs

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.

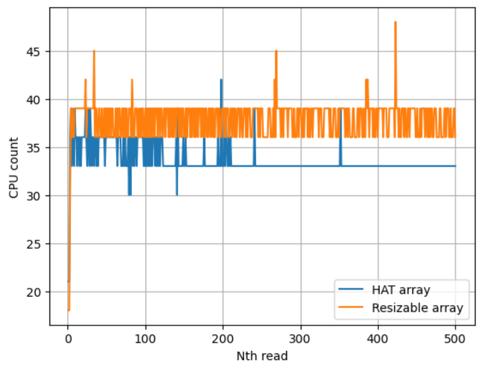


Figure 1.2: Benchmark of scanning through array

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 list performance (question 2)

2.1 Best p for search time complexity

Since skip list bases on probability for randomizing node on each layer. The question is what is the optimal p for skip list in order to make the search fastest. Let's try with the real example of skip list in c++ with p from 0 to 1 (increase by $\frac{1}{8}$).

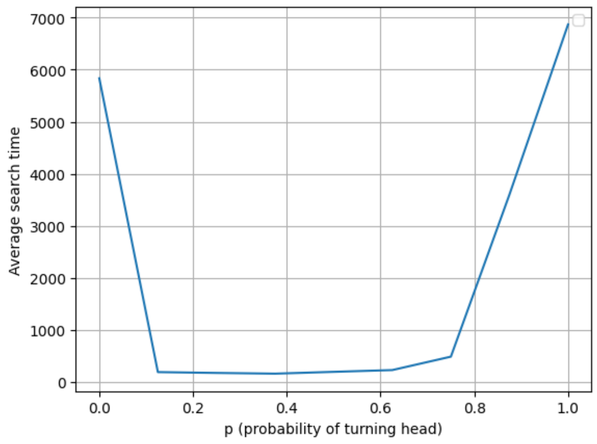


Figure 2.1: Average search time on each probability

From benchmark result, the optimal p is $\frac{1}{4}$ for search time complexity as it takes only 172 cycles on average. The second place is $p = \frac{3}{8}$ which takes 158 cycles on average for searching elements.

Chapter 3

Skip lists

3.1 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.

3.1.1 Append latency

Overall, average cycle count for 10,000 insertions in skip list is 688 cycles while linked list takes 44,836 cycles. This means that skip list insertion is 65x faster than linked list. The following picture shows the cpu cycle on each insert operation on both lists.

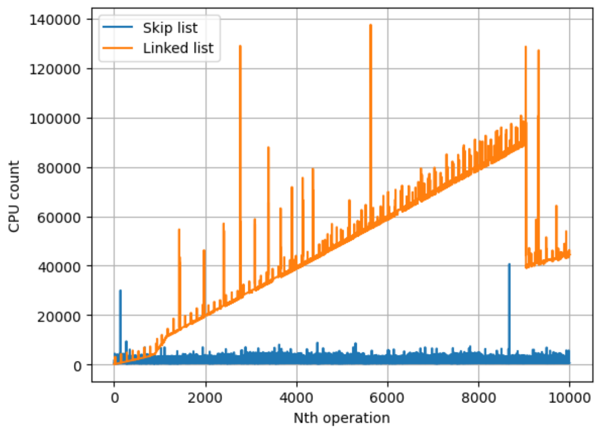


Figure 3.1: CPU cycle per insert operation

As you can see in figure 3.1, skip list insertion is nearly stable at between 400 and 500 cycles with a few strike. On the other hand, since linked list takes $O(n)$ to insert an element, the cycle count is higher as the size of array is increased over each operation.

The result also implies that linked list has a better performance on a small n size while skip list clearly wins the benchmark in a larger scale.

3.1.2 Scan throughput

Overall scan performance goes the same way as the result in section 2.1.1. Skip list takes only 186 cycles on average to search element with 10,000 elements in total while linked list takes 98,651 cycles to search one element. In total, with 10,000 elements, skip list takes 1,860,489 cycles to scan all elements in list while linked list uses up to 985,616,508 cycles. In this competition, skip list clearly wins as it is 530x faster.

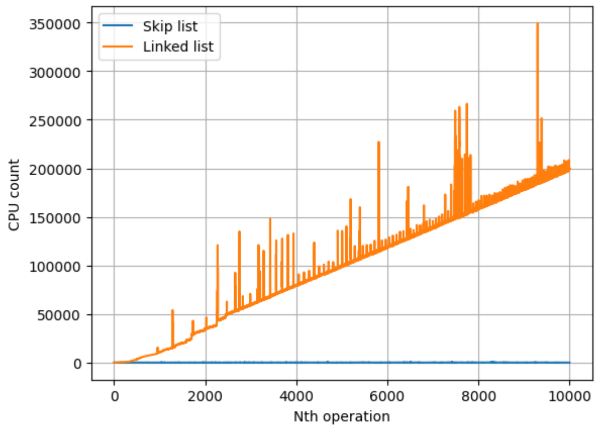


Figure 3.2: CPU cycle per search operation

3.2 Bi-directional search

Searching in one directional skip list takes $O(\log_2 n)$ where n is the amount of element in array since we start from the top layer. Let's improve the running time of search algorithm by allowing it to go up and down. In this way, we look for the layer to start searching, instead of alway start from the highest level. In this way, the search will take $O(log_2 d)$ where d is the amount of elements smaller than the given key because the highest level we can go is $O(log_2 d)$ with high probability. This also means that traversing up and down at $\log_2 d$ level also takes $2\log_2 d$ to the desired key. In total, the search takes $2\log_2 d + 2\log_2 d = O(\log_2 d)$. Let's take a look at pseudo code.

```
search(int element) {
        let x -> header;
        let start_node -> header;
        let layer -> 0;
        -- find start node starting from bottom layer (traverse up)
        for i = 0 -> max level so far {
                if x->forward[i].key < element:
                        start_node = x->forward[i].key;
                        layer = i;
        }
        -- traverse down to desired key
        for i = layer -> 0 (inclusive) {
                while start_node->forward[i] < element:
                start_node = start_node->forward[i];
        };
        -- check final key
        if start_node != NULL and start_node->key == element {
                return start_node.key
        }
        return NULL;
}
```

3.2.1 Benchmark

Let's see the performance difference between one-directional skip list and bi-direction skip list search algorithm. The benchmark is to measure scan throughput of 1,000 sizes with 20 max levels. The result is the following.

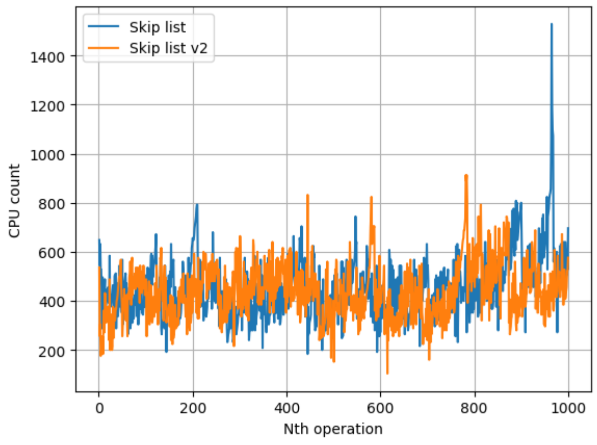


Figure 3.3: CPU cycle per search operation

The performance is slightly different that bi-directional search takes 436 cycles on average while one-directional search takes 451 cycles. In conclusion, bi-directional skip list search is faster by 4%.

Chapter 4

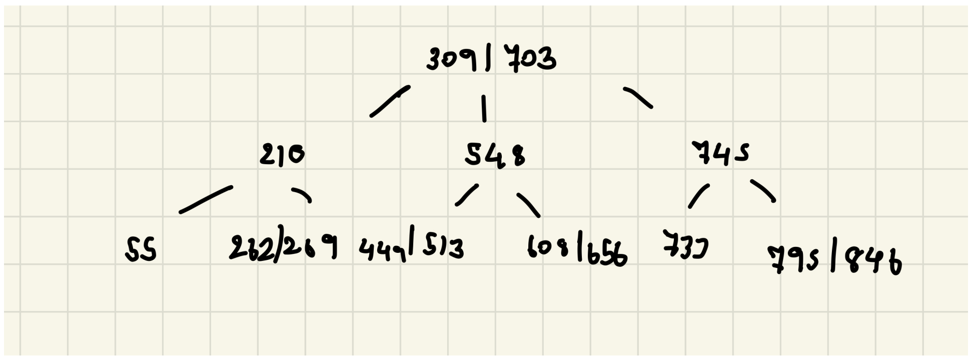
(a,b) tree

4.1 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



4.2 Key deletion

Suppose that we want to delete 309 in *(2,3) tree* (Firgure 4.1), it falls into case 2 which we need to merge from sibling (Figure 4.2).

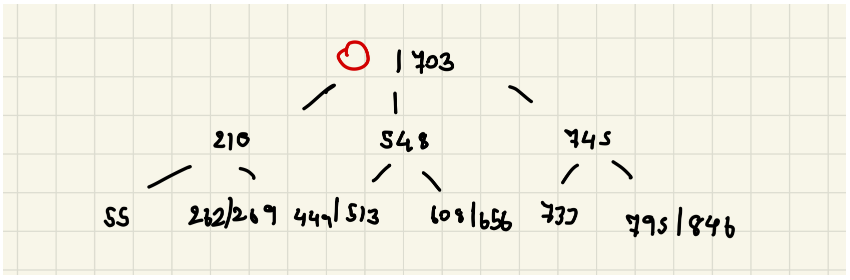


Figure 4.1: Unbalanced tree without key 309

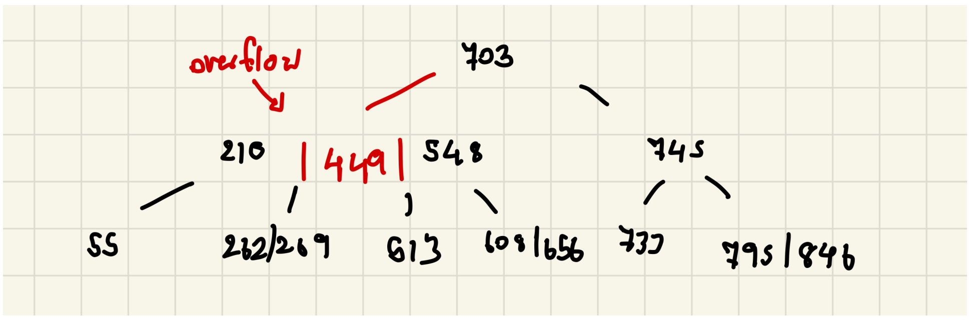


Figure 4.2: Tree after merging

However, the merged tree in figure 4.2 is not balance as the second layer node has 3 keys. So, we need to project 449 up to root.

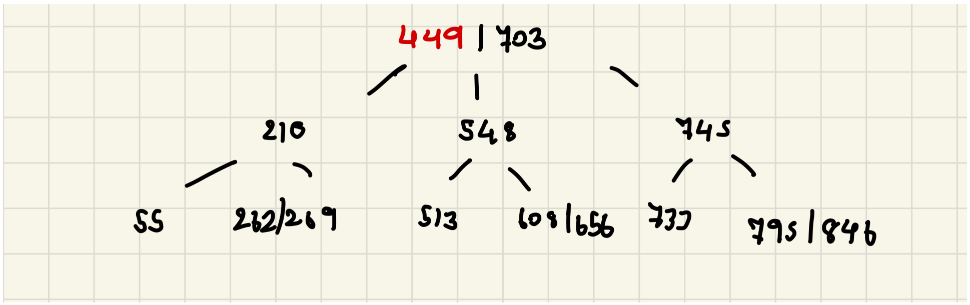


Figure 4.3: Final tree

As you can see in Figure 4.3, the replacements are 449 which is the result from merging non-root tree in figure 4.2.

$$\begin{aligned}\alpha_{\text{me}} &= a - 1\\ \alpha_{\text{sib}} &= a\\ \alpha_{\text{sib}} + \alpha_{\text{me}} &= 2a - 1 \le b\end{aligned}$$

Chapter 5

B-tree speed