An Experimental Study on Treaps

Jiacheng Ye
904973

Experiment Environment.

The study is run in a Macbook Pro (15-inch, 2018) with macOs Catalina. It has 16 GB memory with 2400 MHz DDR4. Its CUP is the Intel Core i7 6 cores with 2.6 GHz. The program is written in C. The compiler gcc is installed by Xcode by default, and its version is 4.2.1.

Data Generation

The data is generated by a separate program, called "data_generator.c". Each data is a pair (id, key), (I.e. A struct in c), where id is a unique identifier and key is the search key of the element. Both id and key are integers. The id is a determined and increasing number, which starts at 1. And the key is a random number that are uniformly generated within [0, $10^{\wedge 7}$]. Each operation, i.e., insertion, deletion and search, is also a pair (type, data).

The program has 4 interfaces. A function called "gen_element()" is used to create an instance of the data, and assign a random search key to it.

Sudo code:

gen_element(int *id_next):

- Create an instance of data element
- x.id <-- id_next;
- x.key <-- generate_random_number($10^{\wedge 7}$);
- Return x;

A function called "gen_insertion(int *id)" is used to create an insertion. It gets a new data element from gen_element(), then assign an id to it, then attach it to an insertion operation, then return the insertion.

Sudo code:

gen_insertion(int *id_next):

- x <-- the new element generated by gen_element();
- Return an insertion of x;

A function called "gen_deletion(int *id_next, Operation *arr, int num)" is used to create a deletion. It randomly chooses an integer between [1, id_next], i.e., it randomly chooses a used id. Then it looks up all the existing operations, to find out if there is a deletion which has an element with the same id. If not, then it will just return the search key of the element with the id. Otherwise, it will draw a random key uniformly from the range [0, $10^{\wedge 7}$], and return a deletion with that key.

Sudo code:

gen_deletion(int *id_next, Operation *arr, int num):

- id_{del} <-- generate_random_number(id_next);
- Create a data element x with id_{del};
- If there is an operation with type DELETE and the element id is the same,
  - key_{del} <-- generate_random_number($10^{\wedge 7}$);

- Assign the key to x;
- Return a deletion of x;

- Otherwise, keep searching in the operation array and after find out x's key, return a deletion of x;

Lastly, a function is called "gen_search()" is used to create a search. It randomly chooses an integer between [0, 10^7], then return a search with that key.

Sudo code:

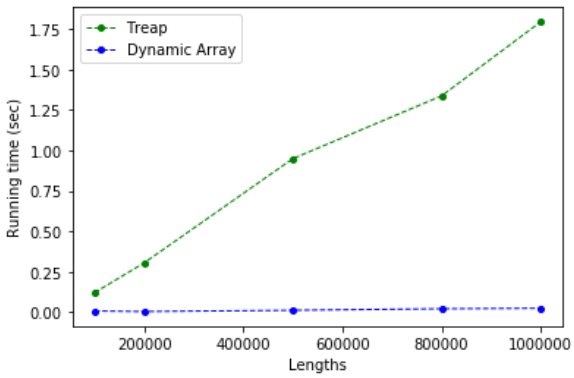
gen_search():

- key_{search} <-- generate_random_number(10^{7});
- Create a data x and assign key_{search} to it;
- Return a search of x;

Experiments

Exp 1: Time v.s. Number of Insertions.

In this experiment, we study the total running time (of the randomized treap and the dynamic array, respectively) v.s. the lengths Lins of insertion-only sequences. The lengths are 0.1M, 0.2M, 0.5M, 0.8M, 1.0M, where M = 10^{6}.

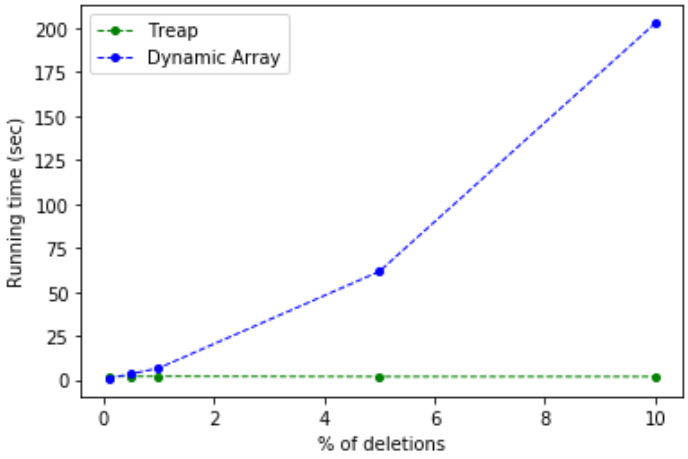


The Treap v.s. Dynamic Array

The graph indicates that the Treap clearly takes more time than the Dynamic Array for insertion. But for the Treap it does not mean that it takes super long to run, as the longest running time is just 1.75 second. Both of the data structure takes linear time to perform. The Dynamic Array performs as intended; in theory it does take linear time to run in the worst cases. The Treap runs slower than the Dynamic Array from the beginning at L = 100000. Then the running time gap grows as the length increases. The reason why the Dynamic Array is faster is because there is an extra step needed for the Treap, the rotation. A worst case for the rotation is O(height(treap)), so more nodes means more rotations. Then the running time of insertion for the Treap grows longer.

Exp 2: Time v.s. Number of Deletions.

In this experiment, we fix the sequence length L to 1M of updates (including both insertions and deletions). Then we vary the rough percentage, %del, of the deletions in the update sequence, where %del = 0.1%, 0.5%, 1%, 5%, 10%. Then we will run both algorithms on the same sequences and measure their running time.

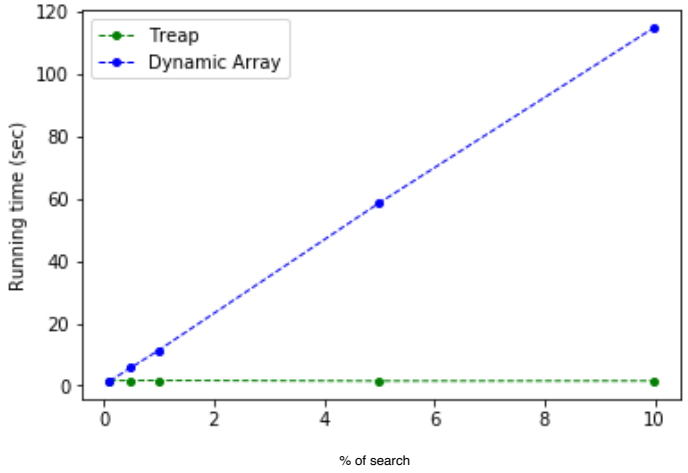


The Treap v.s. Dynamic Array

This graph shows that for deletion, the Dynamic Array is way worse than the Treap. The longest running for the Dynamic Array is even over 200 seconds, while the Treap keeps the running time constant and low. This is quite the opposite of the Exp 1. Also, the Dynamic Array now requires expotential time to perform. This is because the Dynamic Array needs to loops through the entire array until it finds the element to delete. The worst case is O(n). This makes the running time grows super-fast as the number of trails increasing. It also requires to shrink e size to half and copy the whole array if current number of elements is less than ¼ of its size. This operation is quite expensive when the number of elements is large, which is O(n). Thus in total is $O(n^{\wedge 2})$. But on the other hand, the Treap just need to do a binary search (O(logn)) and then rotate the node to a leaf (O(1)) and remove it. Even the worst case is just O(height(treap)). This more advanced look-up functionality gives the Treap a better performance than the Dynamic Array. Also, the time required for the Treap is decreasing as the percentage of deletions increases. Because as it increases, the height and number of nodes decreases, thus, it runs faster.

Exp 3: Time v.s. Number of Searches.

Similar to Exp 2, we use a sequence of a fixed length L = 1M of operations which contains insertions and searches only. We then vary the rough percentage, %sch, of the search operations in the operation sequence, where %sch = 0.1%, 0.5%, 1%, 5%, 10%. Then we will run both algorithms on the same sequences and measure their running time.

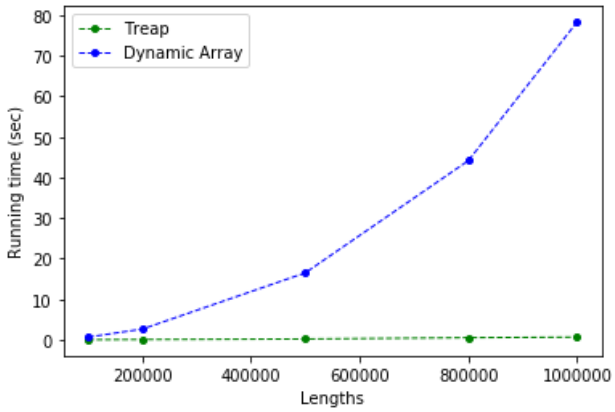


The Treap v.s. Dynamic Array

This graph is not surprising. The Dynamic Array takes linear time to search. And the Treap only takes logrithm time to search, as it is a binary tree. Thus the Treap performs much better in this experiment. But also notice that the Dynamic Array requires more time to delete than search. And it is because of the re-sizing operation. And same situation applies here as well. As the percentage of searches grows, the number of nodes and height decreases in the Treap, thus the line for the Treap is decreasing.

Exp 4: Time v.s. Number of Mixed Operations.

In this experiment, we use operation sequences with mixing insertions, deletions and searches with different lengths L = 0.1M,0.2M,0.5M,0.8M,1M, where M = $10^{\wedge 6}$. Each of these sequences is generated with 5% deletion, 5% search, and 90% insertion. Then we will run both algorithms on the same sequences and measure their running time.



The Treap v.s. Dynamic Array

Overall, we can see that for the Dynamic Array, the operations of deletion and search, especially the deletion, dominant the running time and shows an expotential growth in the running time. (The reason for why is expotential not linear is explained in Exp 2). In total it requires O(n) + $O(n^{\wedge 2})$ + O(n), which is less than $c*O(n^{\wedge 2})$. On the other hand, the Treap's running time is pretty short, due to its capability of fast deletion and search (Both of them only O(log(n))). But it is still increasing, not like in Exp 2/3 that is decreasing. Because the Treap is pretty heavy on insertion. The overall running time for the worst case is O(n) + O(n) + O(n) which is O(n). But as this is a randomized Treap, the probability makes the Treap balanced and in the expeced time, the running time is O(height(treap)), which is O(logn).

Conclustion

From 4 experiemnts, I conclude that the Treap is better when the taks is mixed with different operations like inseration, deletion and search. The more mixing, the better it will perform. But the Dynamic Array may be more suitable for the cases when there is havey insertions only. The Dynamic Array may perform better if combined with hash-maps, as it will save time to look up a particular element.