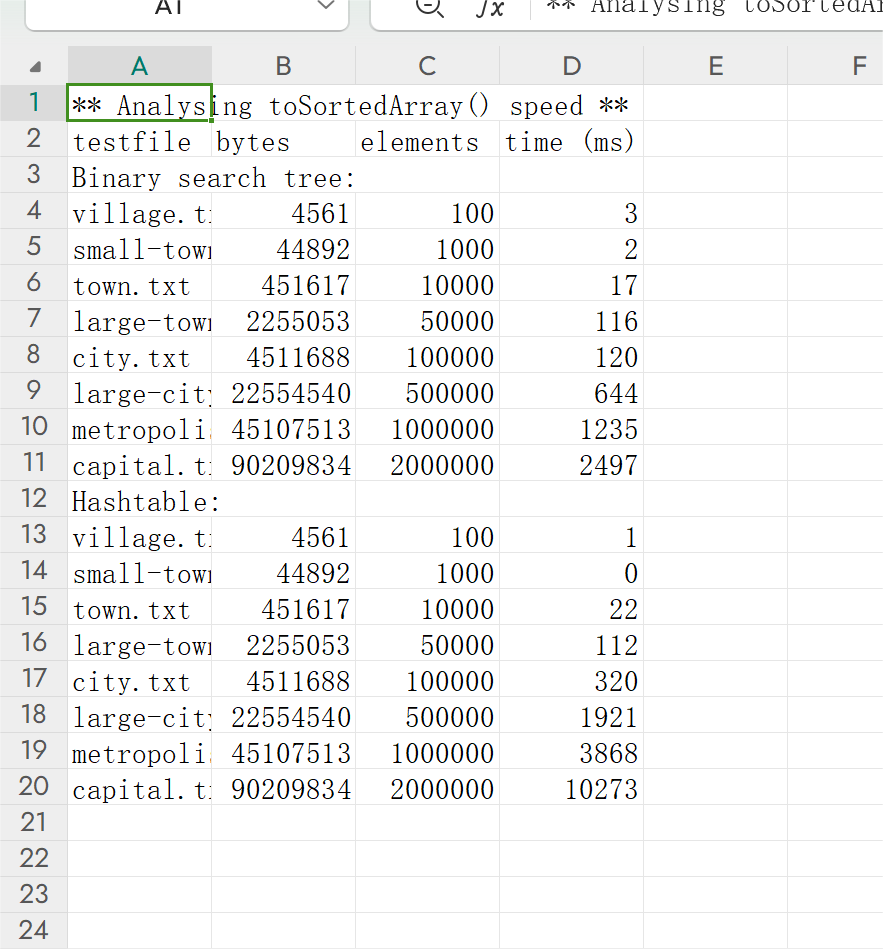
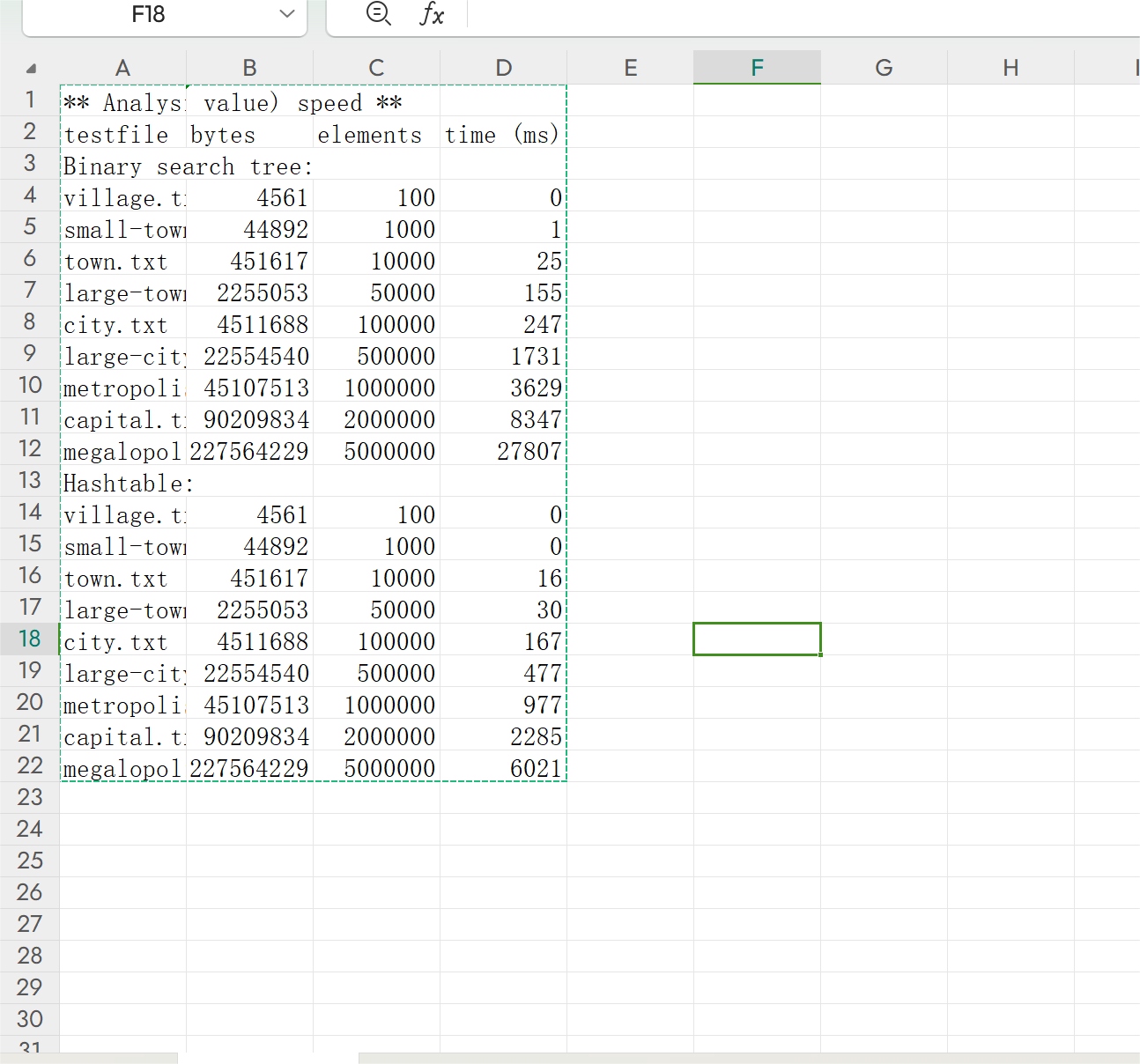
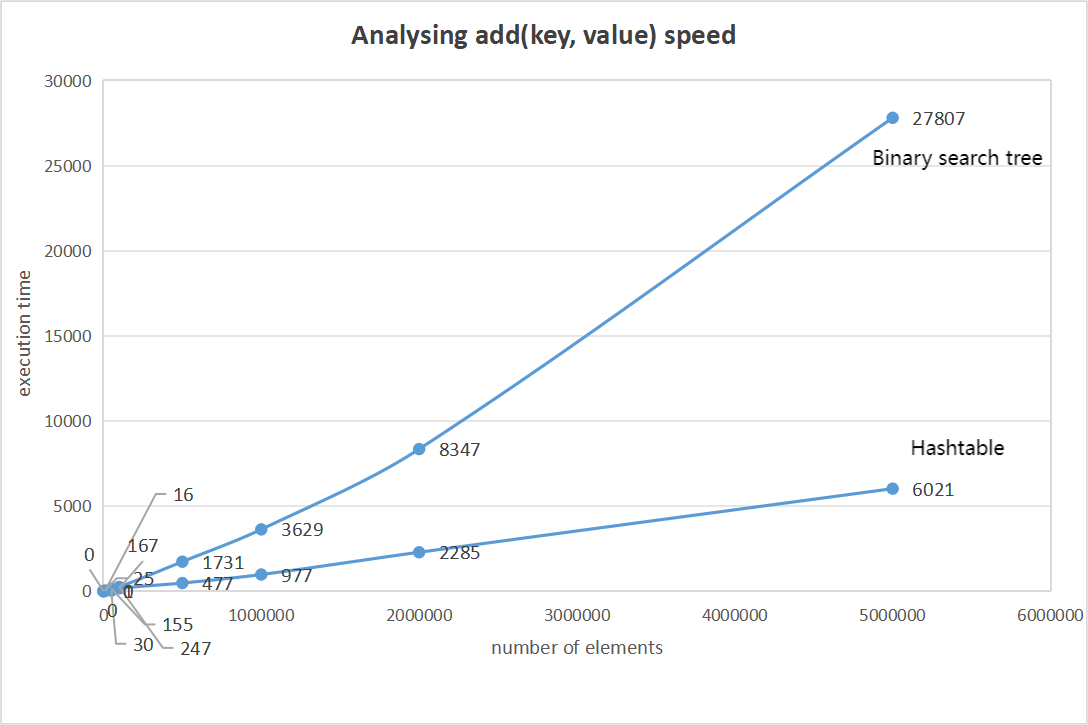
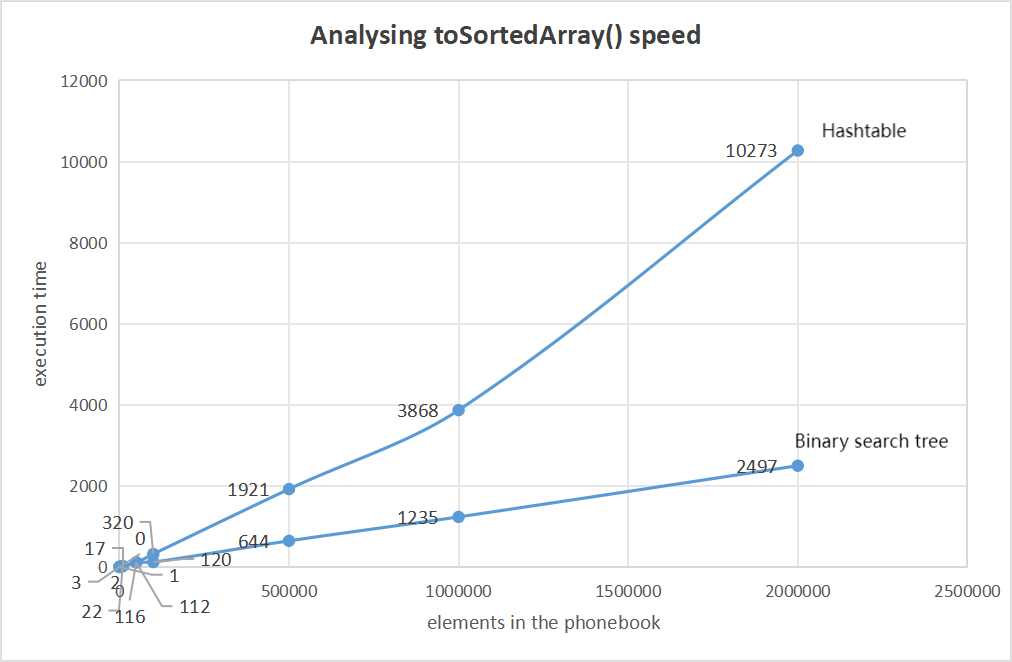
****

****

****

**Which kind of hash functions did you try out for calculating the hashes? Did some functions produce better hashes than others? E.g. more collisions or less collisions in filling the hash table?**

When filling the hash table I tried using multiple hash functions to calculate the hash value. Different hash functions do differ in the number of collisions (collision) in the hash table. Common hash functions include division method, square taking method, folding method, random number method, etc. For example, the division method generates a hash value by taking the residue of a fixed value of the key value, which is simple and easy to implement, but causes collisions if the residue of multiple key values is the same. By taking the middle few bits of the square of the key value as the hash value, this method can reduce the collision probability of some regular key values.

**What fill factors did you use when creating and reallocating the arrays? Was the performance better with some fill factors?**

As for the fill factor, the fill factor I use is LOAD \_ FACTOR, a value of 0.45, which determines when the hash table expands. A lower fill factor means that the hash table expands earlier, which helps to reduce collisions, but increases the space overhead and the frequency of expansion operations. The logic of redistribution in the add method, which doubles the size of the hash table. Can maintain a relatively stable filling factor, while avoiding too frequent capacity expansion operations.

**What was the impact of reallocation on performance with different solutions? Was the table finally having very much empty (null) objects and was thus wasting memory, perhaps too much?**

The effect of redistribution on performance is mainly reflected in the following aspects: Time complexity: The redistribution usually requires the time complexity of O (n), where n is the number of elements in the hash table. This is because you need to recalculate the hashes of all the existing elements and place them in the new array. This causes the program to pause or significantly slow during the realsignment. Memory allocation: Resigning also involves allocating memory to new arrays. This can result in additional memory overhead, especially in memory-constrained environments. In addition, frequent reallocations can cause memory fragmentation problems. Cache efficiency: After the reassignment, all the elements in the hash table are located in the new memory location. This may result in an increased cache unhit rate because the CPU cache may still contain data from the old location. For my code, there is a potential problem that, in the ensureCapacity method, the reassigned array size is calculated by multiplying the loading factor (1.0 + LOAD \_ FACTOR). Here the loading factor is 0.45, but multiplying by (1.0 + LOAD \_ FACTOR) actually causes only a slight increase in capacity. This loading factor (e. g., 0.45) reduces the likelihood of a collision, but increases memory consumption because the array maintains more empty positions.

**How deep did the BST get when filling it with data? Any difference when using different hash functions to generate the hashes (keys for the tree nodes)?**

The binary search tree has a depth of 32. Generating keys using different hash functions may cause the following differences: Uniform distribution: A good hash function can distribute the bonds evenly throughout the bond space, which will help to keep the binary search tree in balance. If the hash function is uniform, then the tree depth may remain relatively low. Aggregation: If the hash function causes the keys to cluster in a specific value or range, then the binary search tree can become very unbalanced. This may lead to a sharp increase in the depth of the tree, especially in the worst case, which may degenerate to a chain table (all the bonds are clustered together). Collision: The hash function may create collisions, where different inputs generate the same hash value (keys). If multiple elements have the same keys, then they will be added to the same location in the tree, potentially overriding the previous value (depending on how your implementation handles insertions with the same keys). This may affect tree shape and depth, especially if the way collisions are handled causes an imbalance in the tree.

**Which techniques for handling collision did you use in both data structures? Did you try out different techniques, and why did you select the one in the implementation?**

The way I handle hash table collisions is the chain address method. In the add method of the hash table, when a collision is found (namely, the key with the same hash value already exists in the table), the code creates a linked table at the same index position, adding a new key value pair to the end of the linked table. If the key already exists, update the corresponding value.

For a binary search tree (BST), the concept of collision does not apply directly, because the BST organizes the data by comparing the values of the keys. BST avoids repeating keys by keeping all of the keys of the left subtree smaller than its parent and all of the keys of the right subtree larger than its parent. If you attempt to insert a node with the same key, BST usually updates the value corresponding to that key.

I choose it for the following reasons:

Flexibility: The chain address method allows hash tables to dynamically handle any number of collisions without changing their size. Link tables can grow with demand without the need to re-hash all elements or move elements.

Easy to implement: the chain address method is relatively easy to implement and understand. Each hash table index is only required to maintain a header pointer to a linked table, and the time complexity of inserting and finding operations is usually only related to the length of the linked table.

Adaptability: Chain address method has good adaptability for uneven bond distribution or hash function is not ideal. Even if the hash function causes a large number of collisions, the chain address method can handle them efficiently.

**When trying out different solutions, did you see significant differences in what was printed out in the getStatus() method? What do you print out there, and does it somehow describe the goodness of these implementations?**

This state information changes significantly when trying out different solutions.For example，“KeyValueArray reallocated 10 times, each time doubles the size ，KeyValueArray fill rate is 60.28% ，Hash table load factor is 0.45，Hash table capacity is 61651 ，Current fill rate is 20.02% ，Hash table had 1506 collisions when filling the hash table. ，Hash table had to probe 0 times in the worst case. Hash table had to reallocate 7 times.Tree has max depth of 0. Longest collision chain in a tree node is 0 ，Min path height to bottom: 5 ，Max path height to bottom: 32 ，Ideal height if balanced: 10.0”

This state information does describe the "good or bad" of these implementations. They provide metrics about data structure performance, memory usage, and balance, such as fewer collisions. The parameters and strategies that help me evaluate and adjust the data structure. By comparing the state information of different implementations, the data structure and configuration of the most suitable application scenario can be selected.

**Which fast sorting algorithm did you implement in 03-binarysearch and used here? Did you try out if some other sorting algorithm could have performed faster with this data set? Would quicksort be slower or faster than heapsort? Did you have to increase the stack or heap size to make this work?**

I implemented the ordinary fast sorting algorithm in 03 binary search but implemented merge sorting here. Ordinary quick sorting causes stack overflow, so change to merge sorting. Inidation and heap sorting are faster.Quicksort and heapsort are both efficient sorting algorithms, but they have different characteristics that can affect their performance in different scenarios. In general, it's not possible to say that one is always faster than the other; it depends on the specific data being sorted, the implementation details, and the system characteristics.

Quicksort is a divide-and-conquer algorithm that selects a pivot element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The algorithm then recursively sorts the sub-arrays. Quicksort's performance can be highly dependent on the choice of pivot, and in the worst case, it can have a time complexity of O(n^2). However, with a good pivot selection strategy (such as median-of-three or random pivot selection), the average case time complexity is O(n log n).

Heapsort uses a binary heap data structure to sort an array. It first builds a max heap from the input array, ensuring that the largest element is at the root. Then, it repeatedly removes the root (which is the largest element) and places it at the end of the array, re-heapifying the remaining elements to maintain the heap property. This process is repeated until the entire array is sorted. Heapsort has a worst-case time complexity of O(n log n) and is generally considered a more stable and predictable algorithm than Quicksort in terms of performance.

Quicksort is a recursive algorithm, so it uses the call stack to keep track of subproblems. If the input size is very large or the recursion depth is high, it may require a larger stack size. On the other hand, heapsort operates entirely within the heap data structure and doesn't require a significant amount of additional stack space. Instead, it uses the heap, which is typically allocated in the main memory.However, I encounter stack overflow errors caused by deep nested recursion in Quicksort. I need to use techniques such as tail recursion optimization or use iterative methods to explicitly manage the stack to optimize the algorithm.