We have allowed you to comment and suggest on the Google Docs (ping us on the message board if you can't) if you would like to provide any feedback (anything small or big such as a typo, grammar mistake, suggested change, etc. is welcome).

CSE 332: Data Structures and Parallelism

P2 Writeup Template

Instructions

Make a COPY of this template (Go to: File -> Make a copy) and answer every question. Export your writeup as a PDF to submit to Gradescope. Only ONE person from your group should submit the writeup to Gradescope.

To receive **FULL CREDIT**, you must:

1. Use this template for your write-up.
2. Select corresponding pages for each question when you are submitting your write-up to Gradescope.
3. Add your partner to your group on Gradescope after you submitted your write-up.

Names: Allan Ji, Casper Zhang

Repo Name: p2-africanelephant

Extra notes

- We recommend that you keep your "N" (as in "N-gram") constant throughout these experiments. (N = 2 and N = 3 are reasonable.)
- You should not need to wait for hours and hours for your program to run in order to answer a question. Do have a large size of inputs, but not so large that you are waiting overnight for your program to run (1,000,000 inputs is likely larger than you need).

(1) Project Experience

What was your favorite part of the project? Why?
My favorite part is AVL-Tree because this part helped me master the construction principles of various methods of AVL-Tree. Also, I think this is the most complicated part. We had a bunch of different kinds of bugs in the finish, which helped me practice how to debug correctly.

What was your least favorite part of the project? Why?
My least favorite part is HashTrieMap because we need to use a class I haven't learned at all in this part. SimpleEntry confused me since I'm not familiar with the generic statement. I have to spend a lot of time learning the material outside of class.

How could the project be improved? Why? Be specific and show an example.
I think we can add some application descriptions of uMessage. For example, provide some examples of how to use uMessage in specific. Because when I finish this part, I don't know how to use uMessage. Secondly, the lack of comments in the code of the uMessage class makes it difficult for us to understand the content of the code when conducting experiments.

Did you enjoy the project? Why or why not?
I enjoyed this project because it strengthened my skills in designing different data structures for dictionaries. Also, the experiments in the Write-up give me a clear understanding of the difference using different dictionaries. I learned a lot from this project.

(2) BST vs. AVLTree

Construct inputs for BST and AVLTree to demonstrate that an AVL Tree is asymptotically better than a Binary Search Tree.

You should construct inputs akin to writing tests in P1. Declare a new BST and AVLTree, then add specific elements of your choosing to the data structures. Make sure you add the same element to each data structure in the same order.

Hint: Use worst-case inputs for a BST to demonstrate this.

Just like before, you MUST use the Writeup Experiments Handout on the website to conduct all these experiments and what to put in the headers below.

Hypothesis

What do you think might happen?

Using worst-case inputs for BST and using the same inputs for AVLTree, AVLTree is asymptotically better than the BST. In the beginning, when the size of the tree is relatively small, the difference between the runtime to construct and find the worst-case element for an AVLTree and BST is very small. But, as the size of trees grows larger, the difference will get larger, and the runtime of AVL will always be less than that of BSTree.

The image of the runtime of AVLTree grows with the size of the tree should fluctuate within a small range; if the asymptotic analysis, it can be seen as a linear function whose image is a line almost parallel to the x-axis.
The image of the runtime of BST grows with the size of the tree should grow with the size. If using the asymptotic analysis, it can be seen as a polynomial fit line.

Why do you think that might happen?

The insert() and find() methods of AVLTree have O(log n) runtime for the worst case due to the balanced restriction, while these two methods of BST have O(n) runtime for the worst case. So, as the tree grows larger, the AVLTree will be asymptotically better than BST.

Methodology

Independent Variable(s)

Size: The size of the tree that we use the worst-case inputs to construct the tree.
The type of Tree: AVLTree or BSTree

Dependent Variable(s)

Runtime to construct/insert method of a tree
Runtime of find(worst-case key) method tree

Procedure

We plan to create an AVLTree and a BSTree using the same size and the same order of inputs to insert. Then, we want to use Timing operations to compare the runtime to construct an AVLTree and a BSTree, which is also the runtime to insert the size number of items. Also, we compare the runtime for both trees (storing the same items) to find a worst-case key.

Part I Construct / Insert runtime:
In detail, we choose the Timing operations to calculate the runtime of constructing AVLTree and BSTree:

1. we test the runtime to construct a BSTree with size = 100. we defined the key-value pair in the dictionary to be Integer-Integer since it is easy for us to use for-loop to construct the tree. Then, since we want **use the worst-case inputs to construct BSTree**, we have to **insert elements by increasing order of key**. So, we used the loop starting with i = 0 to size(exclusive) to insert the element (i,i) and updated loop with i++, making sure the key of element we insert is in order of increasing: (0,0),(1,1),(2,2) … (size-1, size-1). Then, using the Timing operation to test these codes above with NUM_TESTS = 10000, NUM_WARMUP = 10 and record it.

2. we now test the runtime to construct an AVLTree with size = 100. We use **the same way in step 1** to define the data type of dictionary to construct AVLTree (Since **we need to make sure to insert the same elements with the same orders**). Then, using the Timing operation to test constructing code of AVLTree with NUM_TESTS = 10000, NUM_WARMUP = 10 and record it.

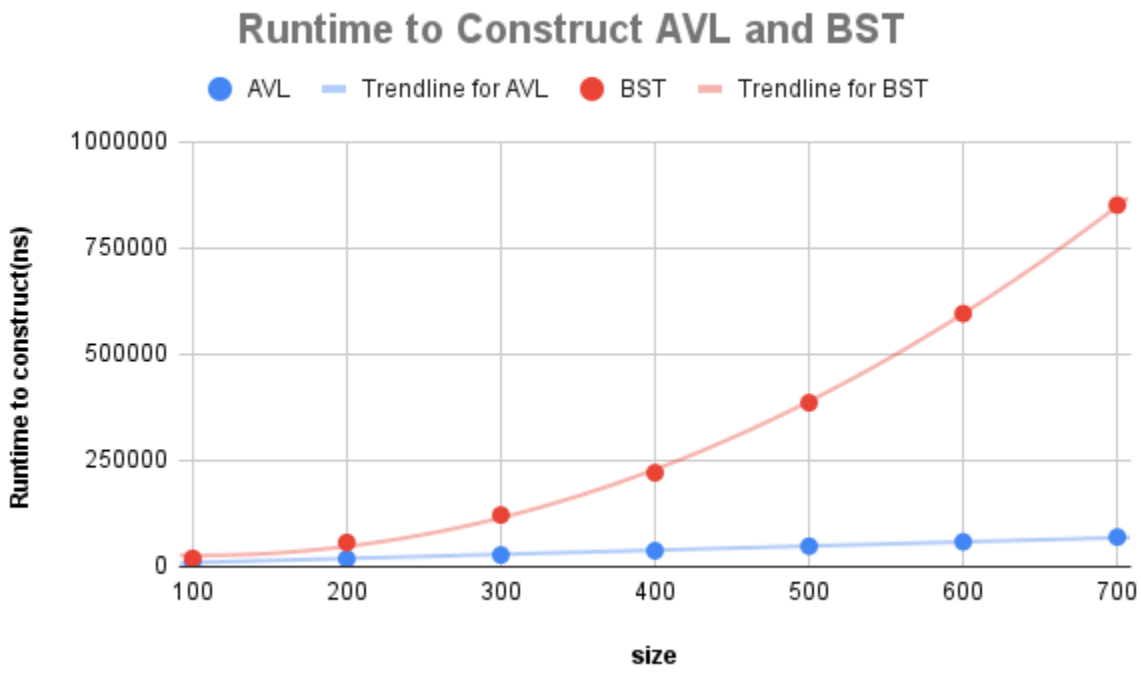
3. We record the runtime of constructing AVL and BST in the line of size = 100. After that, we **repeat Steps 1 and 2 and change the size of the tree** to 200, 300 … 600, and 700 and make the graph of these **7 sets of data**. We found that

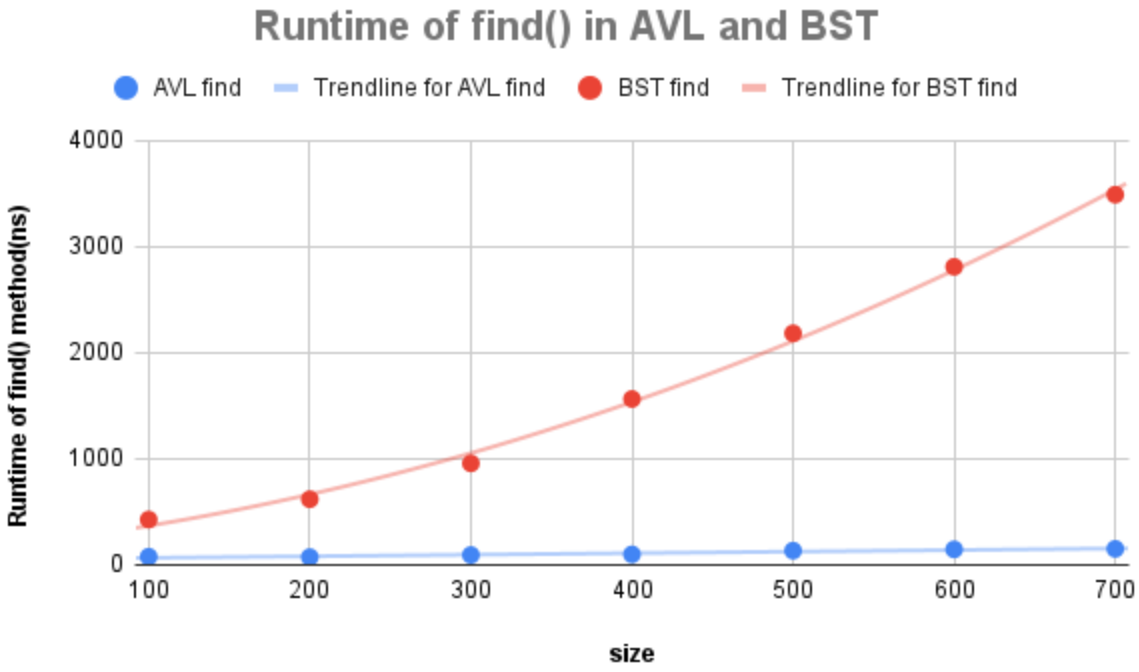
the values of the first set of data are often abnormal, and we guess that it may be caused by too little NUM_WARMUP and too huge NUM_TESTS. After we **increased NUM_WARMUP**, the trend of the data became normal.

Part II: Runtime of find(size):
To examine our comparison, we then used Timing operations to test the runtime of the find methods which find the element in the worst-case:

1. We construct an AVLTree and a BSTree with size = 100 by the same ways in Step 1 and 2 of the part I.
2. We use find() method to **find a key = size** in both AVLTree and BSTree, since the key of items in both trees is 0 to (size - 1). So, find(size) is the worst-case for find() method. Then, we use Timing operations to test the runtime of find with NUM_TESTS = 10000, NUM_WARMUP = 50 and record it, respectively.
3. Repeated Steps 1 and 2 in part II by **increasing the size** to 200, 300, 400 … 700, and 7 sets of data in total to construct AVLTree and BSTree. Then, we collect data and make images of them to see the difference in runtime between AVL and BST.

Results





Conclusion

Interpret your data

In the first image, it shows the comparison of AVLTree and BSTree runtime of constructing(inserting) as the size of the tree change.

Showing by the graph, when the size is small, the runtime of AVLTree and BSTree is almost the same. As the size grows up, the difference between the runtime of constructing two trees becomes greater. The runtime of AVLTree is smaller than that of BSTree as the size growing up.

The trend of the data of AVLTree is **linear**, which can be asymptotically regarded as **a constant line parallel to the x-axis**. The best-fit line of data of BSTree is **polynomial**, meaning the runtime to construct BSTree(insert method) is getting bigger as the size grows.

The second image shows the comparison of AVLTree and BSTree runtime to find the same key as the size of a tree change.

Showing by the graph, the best-fit line of the data of the AVL find method is **linear,** which can be asymptotically regarded as **a constant line parallel to the x-axis**. The

best-fit line of data of BSTree find method is **polynomial**, meaning the runtime to find the same key for BSTree(insert method) is getting bigger as the size grows.

Answer the question in the prompt

Shown by the graph, **the runtime of AVLTree is always less than that of BSTree** in both constructing tree and find method. By analyzing asymptotically, **if the size is to infinity, the runtime of AVLTree will be much less than that of BSTree.**
So, AVL Tree is asymptotically better than a Binary Search Tree.

Was your hypothesis supported or unsupported by the data?

My hypothesis is supported by the data. In both graphs, at first, the difference between the two trend lines is indeed small and grows bigger when the size gets bigger. Also, my hypothesis about the trend of the line is the same as my data interpretation.

Appendix

size	AVL	BST	AVL find	BST find
100	12008	19625	77	427
200	18646	56883	73	619
300	27458	121602	93	957
400	37519	220983	99	1564
500	48125	385833	134	2183
600	58514	595673	146	2810
700	69670	8.51E+05	151	3490
	WARM = 100	TOTAL 10,000		

```
private static int NUM_WARMUP = 100;
private static int NUM_TESTS = 10000;

public static void main(String[] args) throws IOException {

   //CONSTRUCT OR INSERT RUNTIME
   for (int size = 100; size <= 1000; size += 100 ) {
       double totalTimeBST = 0;
       double totalTimeAVL = 0;

       for (int j = 0; j < NUM_TESTS; j++) {
           long startTimeAVL = System.nanoTime();
           AVLTree<Integer, Integer> avl = new AVLTree<>();
           for (int k = 0; k < size; k++) {
               avl.insert(k, k);
           }
           long endTimeAVL = System.nanoTime();


           long startTimeBST = System.nanoTime();
           BinarySearchTree<Integer, Integer> bst = new BinarySearchTree<>();
           for (int k = 0; k < size; k++) {
               bst.insert(k, k);
           }
           long endTimeBST = System.nanoTime();

           if (NUM_WARMUP <= j) {
```

```
                totalTimeAVL += (endTimeAVL - startTimeAVL);
                totalTimeBST += (endTimeBST - startTimeBST);
            }
        }
        double averageRuntimeBST = totalTimeAVL / (NUM_TESTS - NUM_WARMUP);
        double averageRuntimeAVL = totalTimeBST / (NUM_TESTS - NUM_WARMUP);
        System.out.println("Size(x): " + size + " AVL runtime: " +
averageRuntimeBST
                + " BST Runtime: " + averageRuntimeAVL);
    }


    // FIND() RUNTIME
    for (int size = 100; size <= 1000; size += 100) {
        AVLTree<Integer, Integer> avl = new AVLTree<>();
        for (int k = 0; k < size; k++) {
            avl.insert(k, k);
        }

        BinarySearchTree<Integer, Integer> bst = new BinarySearchTree<>();
        for (int k = 0; k < size; k++) {
            bst.insert(k, k);
        }

        double totalTimeBST = 0;
        double totalTimeAVL = 0;

        for (int j = 0; j < NUM_TESTS; j++) {
            long startTimeAVL = System.nanoTime();
            avl.find(size);
            long endTimeAVL = System.nanoTime();


            long startTimeBST = System.nanoTime();
            bst.find(size);
            long endTimeBST = System.nanoTime();

            if (NUM_WARMUP <= j) {
                totalTimeAVL += (endTimeAVL - startTimeAVL);
                totalTimeBST += (endTimeBST - startTimeBST);
            }
        }
        double averageRuntimeBST = totalTimeAVL / (NUM_TESTS - NUM_WARMUP);
        double averageRuntimeAVL = totalTimeBST / (NUM_TESTS - NUM_WARMUP);
        System.out.println("Size(x): " + size + " AVL runtime: " +
```

```
averageRuntimeBST
                + " BST Runtime: " + averageRuntimeAVL);
    }

}
```

(3) ChainingHashTable

Your ChainingHashTable should take as an argument to its constructor the type of "chains" it uses. **Explore which type of chain is (on average, not worst case) asymptotically the best: an MTFList, a BST, or an AVLTree.** Use random inputs (look into Java's Random class to craft random inputs) for each structure (both their find() and insert() methods) here.

Hypothesis

What do you think might happen?

As the file size increases, the time used for constructing the ChainingHashTable, or inserting multiple times, would increase linearly. However, the time used for finding keeps constant.

Moreover, for different types of chain, the construction time varies. Within those three types of chains, MTFList is the slowest, and the AVLTree is the best. Similarly, the finding time varies. Within those three types, MTFList is the slowest, and the AVLTree is the best.

Why do you think that might happen?

When the size grows bigger, the size of the ChainingHashTable would grow greater using the resize function, which would rearrange all items and take much time. However, the time for finding keeps constant since the complexity of find is always O(1) no matter the size of the table. In addition, the number of items stored in a single bucket can be considered to be the same, therefore the finding time for the same algorithm is the same.

Considering the performance of different types of chains, insert time complexity of MTFList is O(N), that of AVLTree is O(logN), and BST is O(N) for worst case, and all insert function are called for n times, while n equals to the number of items in the Chain. However, BST could have an average case better than O(N) since it could be better if the order of inserted items is not sorted. As the time complexity of find, that of HashTable is O(1). After we reach the bucket, AVLTree's find is O(logN), BST's is O(N), and MTFList's is O(N). Similarly, BST's runtime could be faster for the same reason mentioned above.

Methodology

Independent Variable(s)

Size: Number of items in the ChainingHashTable
Chain type: the type of dictionary used to store value in each bucket

Dependent Variable(s)

Insertion time: time used to construct the ChainingHashTable (insert all items) using particular size
Finding time: time used to find the particular values in the ChainingHashTable

Procedure

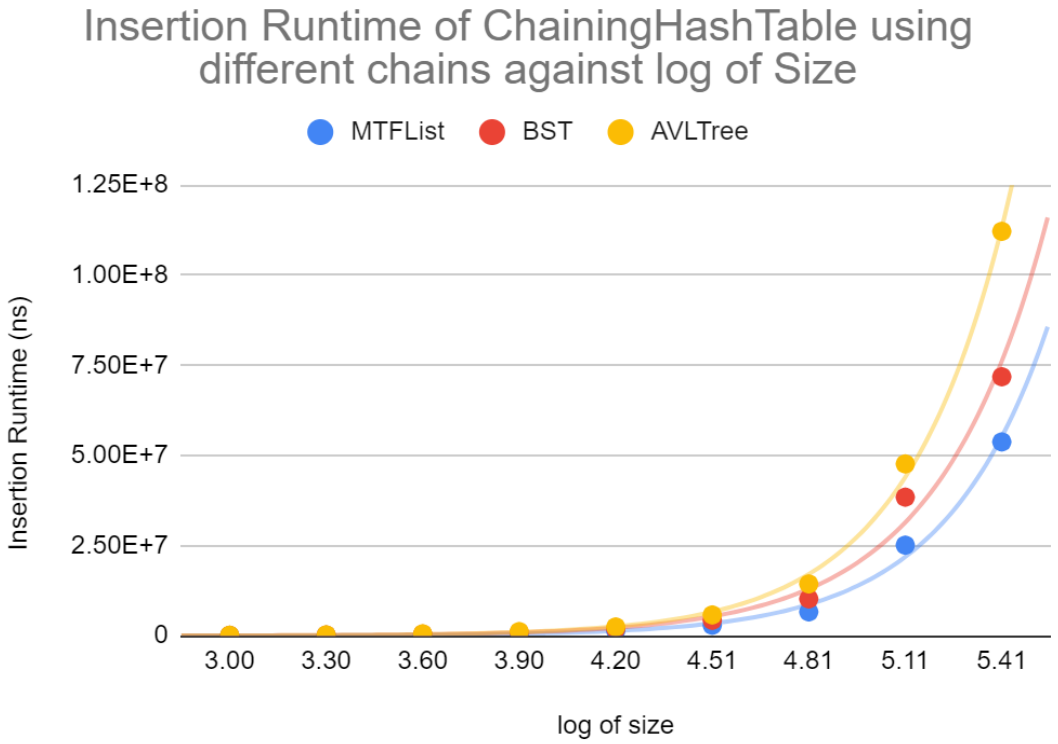
Firstly, we pick **9 numbers of items (size)** to be inserted into the ChainingHashTables using different types of chains, which are 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, and 256000. For example, **if size equals 1000, we construct a ChainingHashTable with 1000 items**. In other words, we call insert 1000 times. We start picking up a relatively large number, and then **double this number each time**. The reason we multiply the size by two each time is that we want to make sure each time the size doubles, one extra resize function is taken place. In this case, we can avoid the case that when size increases, some trials the ChainingHashTable resize but others not.

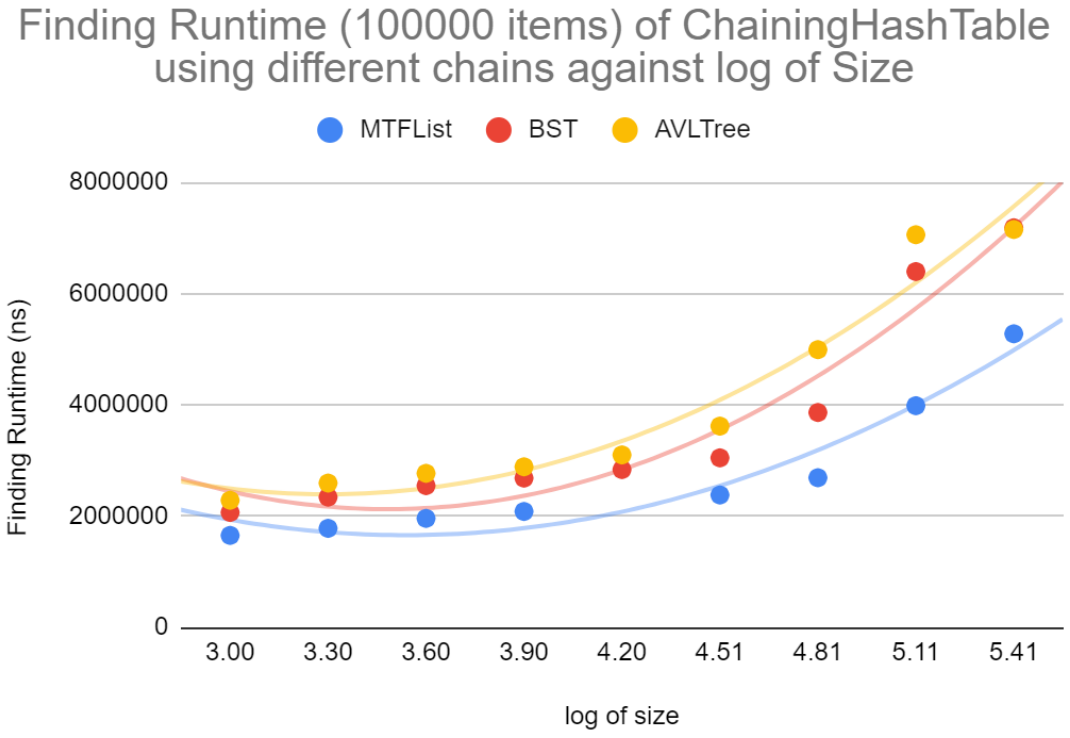
For each trial, we first **generate an array of random numbers with the corresponding size using the random function**. We regenerate the random object each time in order to eliminate the confounding factor of random, for example, some random seed may be too “lucky”. The random number generated is between 0 and INTEGER.MAX_VALUE. Moreover, we construct an array of numbers to be found by random function regardless of whether it exists in the array or not, which is generated by **randomly selecting 100000 values**.

After we have items to be inserted, we can insert those values into the ChainingHashTables using different types of chain, which are MTFList, BST, and AVLTree. Therefore we use the **same array each time to insert value**. We set the **number of tests to be 400**, and the **number of warm-up to be 100**. In another word, we run the code for each type of ChainingHashTable and each size of data for 400 times, and the first 100 runs are considered as warm-up and don’t count to the total. Next, we **start the insert timer and begin to construct the table**, and then add up the time into total. Simultaneously, we **start the find timer and begin to find 100000 random numbers** in the ChainingHashTable, and then add up the time into total. Outside of the loop, we calculate the average construction runtime and average finding runtime.

Finally, we print out the average times in the console and collect those data into the excel.

Results


Insertion Runtime of ChainingHashTable using different chains against log of Size
MTFList
BST
AVLTree
1.25E+8
1.00E+8
7.50E+7
5.00E+7
2.50E+7
0
Insertion Runtime (ns)
3.00
3.30
3.60
3.90
4.20
4.51
4.81
5.11
5.41
log of size


Finding Runtime (100000 items) of ChainingHashTable using different chains against log of Size
MTFList
BST
AVLTree
8000000
6000000
4000000
2000000
0
Finding Runtime (ns)
3.00
3.30
3.60
3.90
4.20
4.51
4.81
5.11
5.41
log of size

Conclusion

Interpret your data

For insertion, the ChainingHashTables using all three types of chains have an exponential runtime. Among those, performance of MTFList is the best, while that of AVLTree is the worst.
For finding, the runtime has polynomial runtimes. Among those, the performance of BST and AVLTree are similar, while MTFList has the fastest runtime.

Answer the question in the prompt

The chain type that is asymptotically the best is the MTFList

Was your hypothesis supported or unsupported by the data?

Our hypothesis is not supported by the data. Contrary to our hypothesis, the MTFList is the best data type. The possible reason might be that AVLTree takes more time on rotating, and BST needs to compare two items at a time to decide to go with which child. Furthermore, since those data structures only store a bucket of data after being hashed inside the ChainingHashTable, the number of items in each dictionary is small. Therefore the advantage of AVLTree and BST is not obvious.
For finding time, MTFList has the fastest speed, while the other two have the similar experience, which also contradicts our hypothesis. The possible reason might be that MTFList have a higher possibility of finding the same value which was moved to the front of the list when the number of items is greater. Moreover, since the AVLTree and BST use the same find algorithm, they are similar in average case. However, the finding time increases when the log of size increases, since the bucket size also increases when size increases, therefore it takes more time to find an element in a larger dictionary.

Appendix

size	log of size	MTFList	BST	AVLTree	MTFList	BST	AVLTree
1000	3.00	74782	109706	154363	1655679	2068567	2286862
2000	3.30	128044	194424	256432	1782793	2343583	2596303
4000	3.60	289373	430799	549327	1961601	2550097	2771181
8000	3.90	621982	923989	1161786	2085947	2685895	2888583

16000	4.20	1308597	1902097	2443175	2273375	2838988	3102204
32000	4.51	2908367	4287507	5774481	2381991	3049715	3619959
64000	4.81	6624501	10227403	14374670	2692443	3864937	4994373
128000	5.11	25127441	38442514	47670343	3986748	6397911	7060012
256000	5.41	53771644	71840861	112181323	5279561	7184959	7154483

```
public class ChainingHashTableExperiment {
   private static int NUM_WARMUP = 100;
   private static int NUM_TESTS = 400;
   private static int[] arr_insert;
   private static int[] arr_find;

   public static void main(String[] args) throws IOException {

       for (int size = 1000; size <= 256000; size *= 2) {

           double totalConstructTimeMTF = 0;
           double totalFindTimeMTF = 0;
           double totalConstructTimeBST = 0;
           double totalFindTimeBST = 0;
           double totalConstructTimeAVL = 0;
           double totalFindTimeAVL = 0;

           int find_size = 100000;

           for (int j = 0; j < NUM_TESTS; j++) {
               arr_insert = new int[size];
               for (int i = 0; i < size; i++) {
                   arr_insert[i] = new
Random().nextInt(Integer.MAX_VALUE);
               }
               arr_find = new int[find_size];
               for (int i = 0; i < find_size; i++) {
                   arr_find[i] =  new
Random().nextInt(Integer.MAX_VALUE);
               }
```

```
                double[] MTF = timeTest(size, new
ChainingHashTable<>(MoveToFrontList::new));
                double[] BST = timeTest(size, new
ChainingHashTable<>(BinarySearchTree::new));
                double[] AVL = timeTest(size, new
ChainingHashTable<>(AVLTree::new));

                if (NUM_WARMUP <= j) {
                    totalConstructTimeMTF += MTF[0];
                    totalFindTimeMTF += MTF[1];
                    totalConstructTimeBST += BST[0];
                    totalFindTimeBST += BST[1];
                    totalConstructTimeAVL += AVL[0];
                    totalFindTimeAVL += AVL[1];
                }
            }
            System.out.println("MTF: " + "size: " + size + "
averageInsertRuntime: " + totalConstructTimeMTF / (NUM_TESTS -
NUM_WARMUP)
                + " averageFindRuntime: " + totalFindTimeMTF /
(NUM_TESTS - NUM_WARMUP));
            System.out.println("BST: " + "size: " + size + "
averageInsertRuntime: " + totalConstructTimeBST / (NUM_TESTS -
NUM_WARMUP)
                    + " averageFindRuntime: " + totalFindTimeBST
/(NUM_TESTS - NUM_WARMUP));
            System.out.println("AVL: " + "size: " + size + "
averageInsertRuntime: " + totalConstructTimeAVL / (NUM_TESTS -
NUM_WARMUP)
                    + " averageFindRuntime: " + totalFindTimeAVL /
(NUM_TESTS - NUM_WARMUP));

        }

    }

    private static double[] timeTest(int size,
ChainingHashTable<Integer, Integer> table) {
        long startTimeInsert = System.nanoTime();
```

```
        for (int j = 0; j < arr_insert.length; j++) {
            table.insert(arr_insert[j], arr_insert[j]);
        }
        long endTimeInsert = System.nanoTime();

        long startTimeFind = System.nanoTime();
        for (int k = 0; k < arr_find.length; k++) {
            table.find(arr_find[k]);
        }
        long endTimeFind = System.nanoTime();
        return new double[]{endTimeInsert - startTimeInsert,
endTimeFind - startTimeFind};
    }
}
```

(4) Hash Functions

Write a new but bad hash function for your CircularArrayFIFOQueue. To exaggerate the difference between the two hash functions, you will want to compare a very simple hash function with a decent one. For this experiment, please include the code in your methodology. You should also briefly describe your new hash function in the methodology.

Asymptotically explore the runtime of your ChainingHashTable when the hash function is varied (on average, not worst case). Use random inputs again.

You should keep all other inputs (e.g. the chain type) constant. You should use the chain you found to be the best (or if you found the ChainingHashTable experiment inconclusive, pick whichever you want). You **may** find `AlphabeticString` useful for this experiment.

Hypothesis

What do you think might happen?

The runtime of ChainingHashTable using decent hash is asymptotically less than using simple hash as size growing to infinity for both insert() and find(). In beginning, the difference between them will be little, but if we think size goes to infinity, the difference will be much greater.

Since we want to consider the variables in x-axis to have linear numeric intervals but the size is almost doubled prime in the resize of ChainingHashTable. The runtime of construct/insert by using simple hash and log size will be **exponential**. The runtime of find by using simple hash and log size will also be **exponential**.

The trend line of the relationship between the runtime of construct/insert by using decent hash and log size will be **linear(almost a constant line parallel to x-axis by asymptotic expectation if size goes infinity)**, and the trend line of the relationship between the runtime of find using decent hash and log size will be **linear(almost a constant line parallel to x-axis by asymptotic expectation if size goes infinity)**.

Why do you think that might happen?

The simple hash function will cause **more collisions** than decent hash functions when we insert new items or find a key. So, if the number of items already in the ChainingHashTable is huge, the collisions of the simple hash function will be more. That means many items will get into the same buckets if using simple hash function comparing using decent hash. So, by decent hash, we can using O(1) runtime to easy get the position of the key in the array. In contrast, for the simple hash, when we get in to the index of the position of the key we find, we have to using O(log n) to find this key in the chain/bucket(Since we will use AVLTree as the type of chain). Thus, using simple hash function will need more time to use insert and find method for ChainingHashTable, and the trend line will be varied from that of using decent hash functions.

Methodology

Independent Variable(s)

Size: The number of items stored in ChainingHashTable
The way of Hash function: how to hash in the CircularArrayFIFOQueue, simple or decent.
CONTROL Variable: the type of chain in ChainingHashTable

Dependent Variable(s)

Average runtime of construct a ChainingHashTable with different hash functions.
Average runtime of find a random key for ChainingHashTable with different hash functions.

Procedure

Part I: Average runtime for construct ChainingHashTable by different hash function:
We first need to make sure the hash function of CircularArrayFIFOQueue is decent hash function now.

1. Initialize the <key, value> type in the dictionary of ChainingHashTable by **<AlphabeticString, Integer>** since we can construct an AlphabeticString using CircularArrayFIFOQueue. We choose the value type to be Integer and all of the value is 0 which is easy to write code since the value doesn't matter of the runtime.
2. We first construct a new empty ChainingHashTable in the first test.
3. We use for-loop from 0 to 100 to create a **ChainingHashTable having 100 items(size = 100) and using AVL as the chain.**
4. In each time of this loop, we first construct an empty CircularArrayFIFOQueue storing char type values and prepare to use it to construct an AlphabeticString.
5. We use another inner for-loop from 0 to 300. In each time of the inner loop, **we first create a Random r, and use it to create a random int in the bound of 26(since there are 26 Alphabets) and add it to 'a'. Then we cast this result to Char, which is a random Alphabetic Char.** Add this char to the CircularArrayFIFOQueue that we already constructed in the outer loop. By repeating this operation **300 times** in the loop, we get **a CircularArrayFIFOQueue with a fixed length = 300**(300 chars in this CircularArrayFIFOQueue).
6. Now, we finished the inner loop. In the outer loop, we continually construct an AlphabeticString and pass the CircularArrayFIFOQueue we just create as a

parameter. This AlphabeticString is random since CircularArrayFIFOQueue is random.

(Because we need to get the average runtime, so the key we insert has to be random, and that is why we create a random CircularArrayFIFOQueue in step 5)

7. Using this AlphabeticString as the key and value 0 as an item, **insert this item to the ChainingHashTable(we created in step 2)**
8. Repeat Step 4 to 7 for 100 times(Size = 100). Then, we construct a **ChainingHashTable having 100 items with a random key**.
9. Set NUM_TESTS = 20, NUM_WARMUP = 3 and use Timing operation to test the code from step 2 to 8 to record the data.
10. Double the size of ChainingHashTable we construct to **200, 400, 800, 1600, 3200, 6400, 12800, 8 sets of data in total.** Repeat the tests steps above.

```
public int hashCode() {
   int c = 0;
   for (int i = start; i < size() + start; i++) {
       c *= 37;
       if (arr[i] != null) {
           c += arr[i].hashCode() * i + arr.length * start * 73;
       }
   }
   return c;
}
```

```
public static void main (String[] args) throws IOException {
   int NUM_WARMUP = 5;
   int NUM_TESTS = 30;
   double HashTime = 0;

   for (int size = 100; size <= 12800; size *= 2) {

       for (int i = 0; i < NUM_TESTS; i++) {

           ChainingHashTable<AlphabeticString, Integer> cht = new
ChainingHashTable(AVLTree::new);

           // Average-Case of ChainHashTable for inserting elements:
           long startTime = System.nanoTime();
           for (int j = 0; j < size; j++) {

               // Create a new random CircularArray with size:
               CircularArrayFIFOQueue q = new
CircularArrayFIFOQueue<Character>(300);
               for (int k = 0; k < 300; k++) {
                   // Get a random char:
```

```
                    Random r = new Random();
                    char c = (char) (r.nextInt(26) + 'a');
                    q.add(c);
                }

                // Create a random AlphabeticString as a key:
                AlphabeticString str = new AlphabeticString(q);
                cht.insert(str, 0);
            }
            long endTime = System.nanoTime();

            if (NUM_WARMUP <= i) {
                HashTime += (endTime - startTime);
            }
        }

        double averageDecentRuntime = HashTime / (NUM_TESTS - NUM_WARMUP);
        System.out.println("Size(x) " + size + "    Run time of hash function: "
                + averageDecentRuntime);
    }
}
```

Comment the old hash function in CircularArrayFIFOQueue and add a new simple hash function(don't add a new another hash method directly), which just return a number 17.Then, Repeat the same operation from step 2 to 10.

```
    public int hashCode() {
        return 17;
//        int c = 0;
//        for (int i = start; i < size() + start; i++) {
//            c *= 37;
//            if (arr[i] != null) {
//                c += arr[i].hashCode() * i + arr.length * start * 73;
//            }
//        }
//        return c;
    }
```

Part II: Average runtime for the find method of ChainingHashTable by using different hash functions.

We first need to make sure the hash function of CircularArrayFIFOQueue is **decent hash function now.**

1. We construct a ChainingHashTable with size = 100 by the same ways in part I above. All items have random keys to make sure we test the average runtime.
2. Use the same way of part I step 4-6 to create a new random AlphabeticString, with **the same fixed length = 300**, as a key.
3. Use the find method of ChainingHashTable we create in step 1 to find the AlphabeticString we create in step 2.
4. Setting **NUM_TESTS = 100000, NUM_WARMUP = 500**, we use Timing operations to test the average runtime of find methods of the code just step 2 and 3, which means **we only test the operation of finding a random key in the same ChainingHashTable** for NUM_TESTS times then records data.
5. **Double the size** of ChainingHashTable to **200, 400, 800,1600, 3200, 6400, 12800, 8 sets of data in total.** Repeat the test steps above.

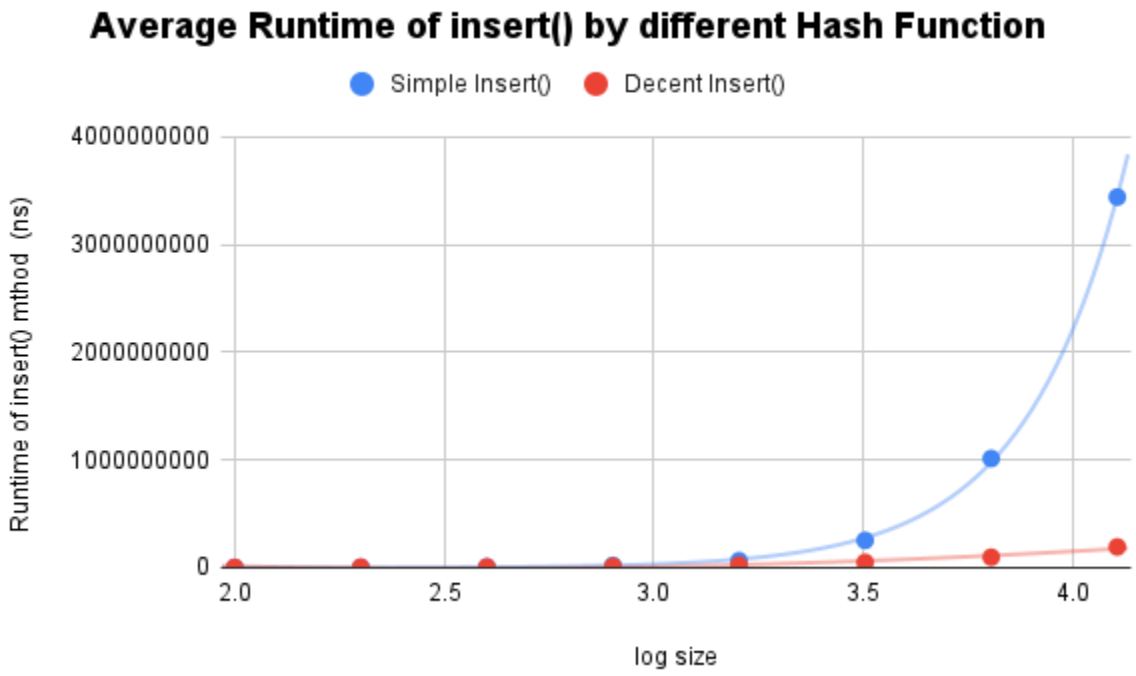
```
public static void main (String[] args) throws IOException {
   int NUM_WARMUP = 500;
   int NUM_TESTS = 100000;
   double HashTime = 0;

   for (int size = 100; size <= 12800; size *= 2) {

       // Runtime of find method:
       ChainingHashTable<AlphabeticString, Integer> cht = new
ChainingHashTable(BinarySearchTree::new);

       for (int j = 0; j < size; j++) {

           // Create a new random CircularArray by given size:
           CircularArrayFIFOQueue q = new
CircularArrayFIFOQueue<Character>(300);

           for (int k = 0; k < 300; k++) {
               // Get a random char:
               Random r = new Random();
               char c = (char) (r.nextInt(26) + 'a');
               q.add(c);
           }

               // Create a random AlphabeticString:
           AlphabeticString str = new AlphabeticString(q);
           cht.insert(str, 0);
```

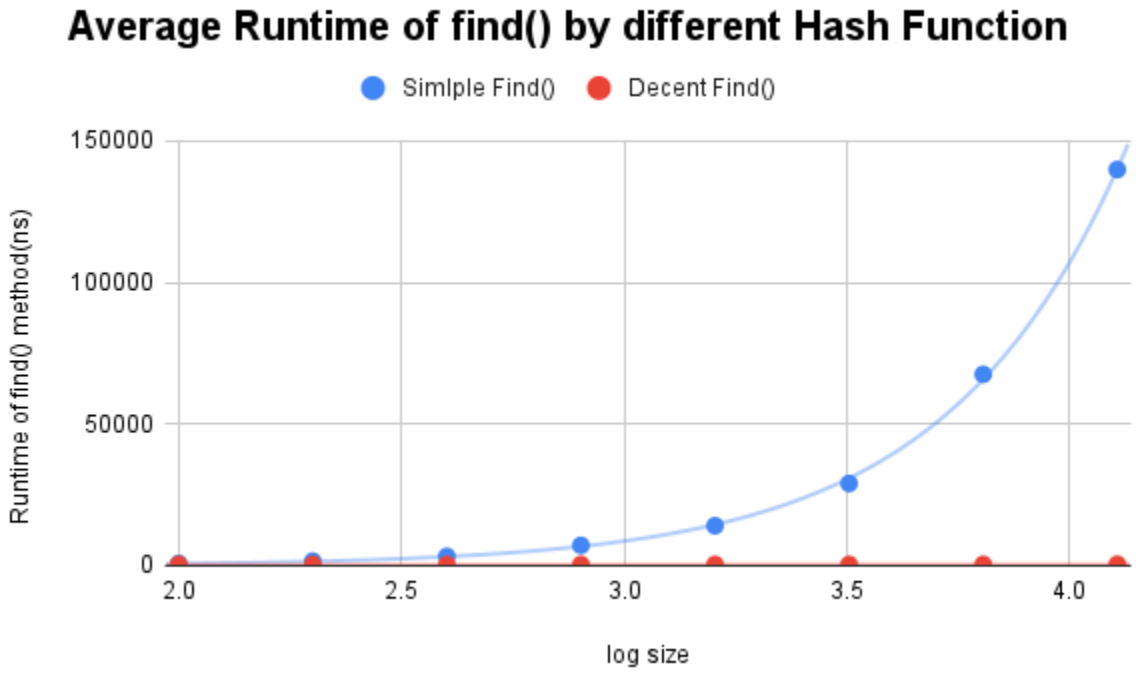
```
            }

        // Starting test the runtime of find:
        for (int i = 0; i < NUM_TESTS; i++) {

            // Create a random str as a random key with the same fixed
length = 300
            CircularArrayFIFOQueue newq = new
CircularArrayFIFOQueue<Character>(300);
            for (int k = 0; k < 300; k++) {
                // Get a random char:
                Random r = new Random();
                char c = (char) (r.nextInt(26) + 'a');
                newq.add(c);
            }
            AlphabeticString key = new AlphabeticString(newq);

            long startTime = System.nanoTime();
            cht.find(key);
            long endTime = System.nanoTime();

            if (NUM_WARMUP <= i) {
                HashTime += (endTime - startTime);
            }
        }

        double averageDecentRuntime = HashTime / (NUM_TESTS -
NUM_WARMUP);
        System.out.println("Size(x) " + size + "    Run time of Simple
Hashcode "
                + averageDecentRuntime);
}
```

Comment the old hash function in CircularArrayFIFOQueue and add a new simple hash function(don't add a new another hash method directly), which just return a number 17**(same with part I)**.Then, Repeat the same operation from steps 1 to 5.

When writing the graph, we **log the size of ChainingHashTable to get linear numeric intervals.**

Results


Average Runtime of insert() by different Hash Function
Simple Insert()
Decent Insert()
Runtime of insert() mthod (ns)
4000000000
3000000000
2000000000
1000000000
0
2.0
2.5
3.0
3.5
4.0
log size


Average Runtime of find() by different Hash Function
Simlple Find()
Decent Find()
Runtime of find() method(ns)
150000
100000
50000
0
2.0
2.5
3.0
3.5
4.0
log size

Conclusion

Interpret your data

First graph is for **the constructing/inserting runtime vs. log size**.
For the Simple Hash Function, the trend line is **exponential** by asymptotic expectation.
For the Decent Hash Function, the trend line is **linear(almost a constant line parallel to the x-axis by asymptotic expectation)**.

The second graph is about **the finding runtime vs. log size**.
For the Simple Hash Function, the trend line is **exponential** by asymptotic expectation.
For the Decent Hash Function, the trend line is **linear(almost a constant line parallel to the x-axis by asymptotic expectation)**.

Also, at the beginning on both graph, when the log size(size) is relatively small, the runtime difference is not obvious. However, if we asymptotically analyze the data(log size goes to infinity) the difference will be huge and the trend of lines are more clear.

Answer the question in the prompt

When we use a very simple hash function in CircularArrayFIFOQueue, the runtime of ChainingHashTable grows exponentially for the methods of insert and find on average cases(relationship with log size). On the contrary, if we use the decent hash function, the runtime of ChainingHashTable grows much slower than that of using simple one. By asymptotically analyzing, the difference between them will be greater and greater if the size is infinity.

Was your hypothesis supported or unsupported by the data?

My hypothesis is supported by the data. The runtime of using simple hash function is indeed much greater than that of using decent hash function. We can see that the blue trend line(simple hash) is always above the red trend line(decent hash), and the difference between them is growing as log size increasing. Considering we use the log size rather than size, the trend line for simple hash becomes exponential which is the same as my expectation in hypothesis.

Appendix

Size	log size	Simple insert	Decent insert
100	2	2353582	2249505
200	2.301029996	4106147	4711300
400	2.602059991	7.82E+06	6.05E+06
800	2.903089987	2.07E+07	1.21E+07
1600	3.204119983	6.48E+07	2.44E+07
3200	3.505149978	2.53E+08	4.92E+07
6400	3.806179974	1.01E+09	9.71E+07
12800	4.10720997	3.44E+09	1.93E+08
		3/20	

Size	log size	Simlple Find()	Decent Find()
100	2	773	437
200	2.301029996	1610	436
400	2.602059991	3286	409
800	2.903089987	7173	428
1600	3.204119983	14104	452
3200	3.505149978	29019	446
6400	3.806179974	67548	455
12800	4.10720997	139852	509
		500/100000	

(5) General Purpose Dictionary

Determine which type of dictionary is the best (BST, AVLTree, ChainingHashTable, and HashTrieMap) using alice.txt as input.

For this experiment specifically, you should not be using a scatter plot and can use a bar graph. As such, this experiment will be much simpler to conduct as there are **no asymptotic** expectations since you are using a fixed text file. How you choose to structure your experiment and use the input is up to you but we expect you to have a **comprehensive discussion** of your results in the conclusion. We leave many things up to your own interpretation e.g. you define what "best" is.

Hypothesis

What do you think might happen?

We think the ChainingHashTable is the best, HashTrieMap is the second best, and BST and AVLTree are similar and are the worst in general case, but BST would be way slower if the input data is sorted.

Why do you think that might happen?

ChainingHashTable is the best since the hash function generally has good performance, since the complexity of insert and find are O(1) in the hash function, and every bucket only stores a small number of items. Moreover, it uses MTFList as Chain, which we found is the best in part 3.
HashTrieMap is the second best since it is generally a good structure to store string-like items since many items may share a part of nodes if they have the same prefix.
AVLTree and BST are similar in average cases if the data passed in is not sorted, and they are worse than the other two algorithms by complexity. However, if the data is sorted, BST would be greatly unbalanced therefore have the worst performance.

Methodology

Independent Variable(s)

Dictionary Type: BST/AVLTree/ChainingHashTable/HashTrieMap
Input: how the data is passed into the function

Dependent Variable(s)

Insertion Runtime: time used to construct the particular type of dictionary (insert all items)
Finding Runtime: time used to find the particular values in the dictionary

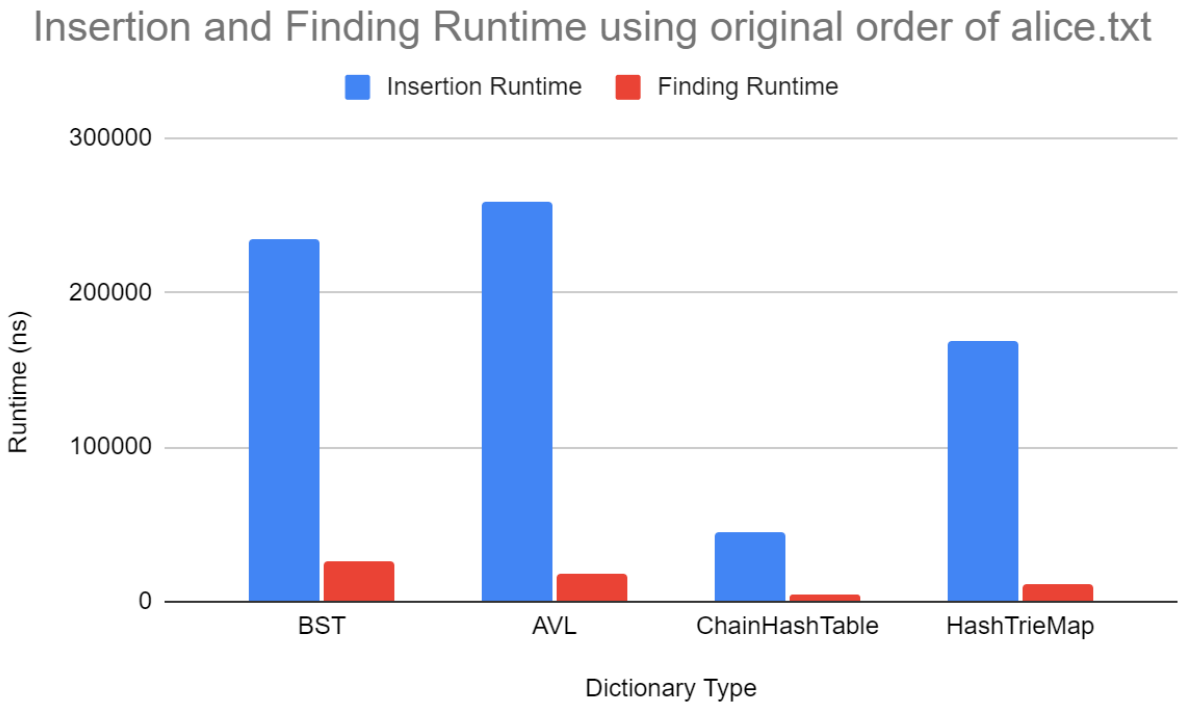
Procedure

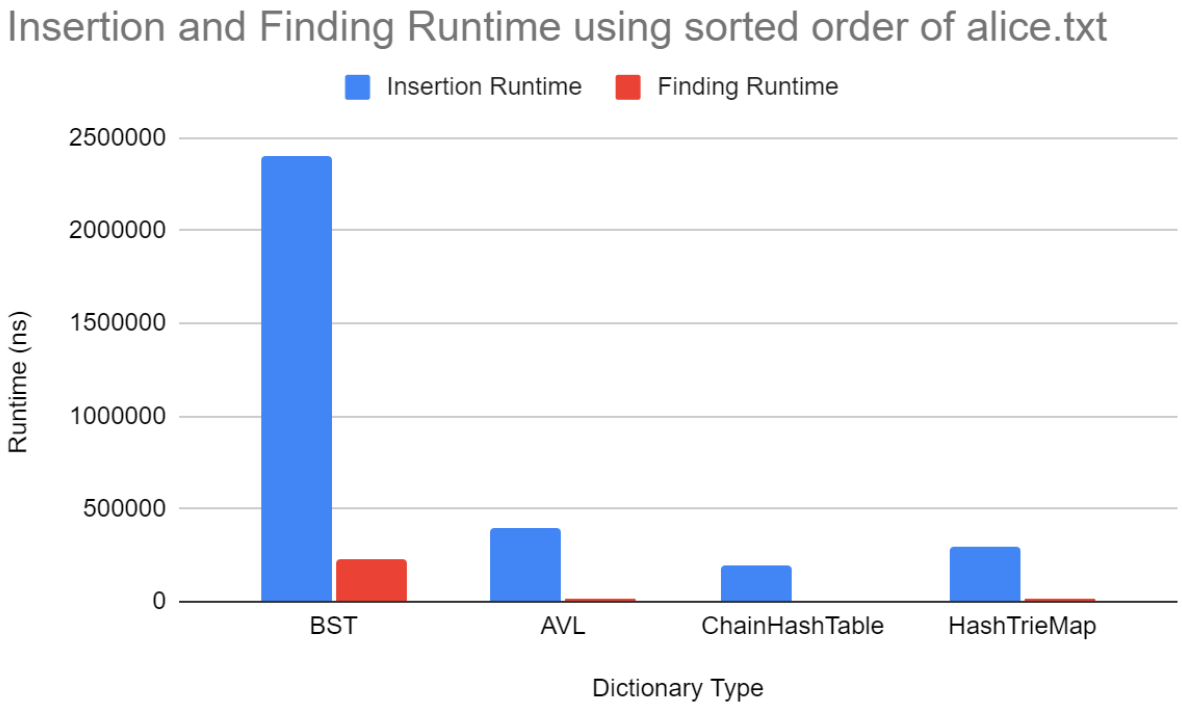
We plan to use two forms of alice.txt as input, one in the original order and the other one in the sorted order. For the original order, we first construct an ArrayList and add all items by order in the txt file into the list. For the sorted order, we create a heap using priority queue, where the comparator uses the compareTo function of AlphabeticString so that when we poll items from the heap, the items are sorted in order.

To measure the insertion runtime, we create a type of dictionary and start an insertion timer for each trial. Then, we insert all items into that dictionary and record the time used for construction. We set the NUM_TESTS to be 20000 and NUM_WARMUP to be 3000. Therefore, we add up the time used for construction for the last 17000 times and take the average, while the first 3000 times are considered to be a warm-up and don't count into total. And we repeat this process for both using the original input and the sorted input.

To measure the finding runtime, we first insert the items as mentioned above. Then, we created a random object with seed equals to the current run number. For instance, it is the 100th run, the random seed is 100. We do that in order to make sure all four dictionaries use the same random object and find the same items. Then, we randomly select an indice from the list and find the following 100 items from that indice. Similar to measurement of insertion, we repeat the experiment for 20000 times, where the first 3000 runs are considered to be warm-up. And we repeat this process for both using the original input and the sorted input.

Results


Insertion and Finding Runtime using original order of alice.txt
Insertion Runtime
Finding Runtime
300000
200000
100000
0
Runtime (ns)
BST
AVL
ChainHashTable
HashTrieMap
Dictionary Type


Insertion and Finding Runtime using sorted order of alice.txt
Insertion Runtime
Finding Runtime
2500000
2000000
1500000
1000000
500000
0
Runtime (ns)
BST
AVL
ChainHashTable
HashTrieMap
Dictionary Type

Conclusion

Interpret your data (i.e. which data structure was "best")

ChainingHashTable is the best. No matter whether the input data is sorted or not, both the runtime of insertion and find are the fastest among those four data structures. For the same reason, the HashTrieMap is the second best. If the input data is not sorted, the performance of BST and AVL is similar. If the input data is sorted, BST is the worst.

Discuss the question in the prompt

We think ChainingHashTable is the best choice. Although this data structure may take up more storage than the other and some buckets may not be filled and it is difficult to implement, it has constant insertion and finding time because of the hash function. Therefore, as we can see in the graph, the runtime is much faster than the others.
HashTrieMap is the second best. When it is used to store the string-like data, it traverses through the trie by the characters instead of comparing each item using the compareTo function as BST and AVLTree. Moreover, it also saves space since some items may share nodes if they have the same prefix. And the data shows that the insertion and finding time is the second best too.
BST and AVLTree are the worst. If the data is not sorted, their performance is similar in average case, since the complexity of insertion is both O(logn) and the complexity of find is O(logn). And AVLTree may take more time on rotating. However, if the data is sorted, the BST is greatly unbalanced, which is like a stick, therefore the runtime would be great, which matches with its worst case insertion time O(n). Due to its unbalanced structure, the finding complexity is also O(n). At contrast, since AVL is always balanced, its performance is about the same in two situations

Was your hypothesis supported or unsupported by the data?

Yes. The result perfectly matches our hypothesis.

Appendix

Unsorted	BST	AVL	ChainHashTable	HashTrieMap
insert	234596.9421	258875.1526	44468	168706
find	25973	17684	4619	11904

Sorted	BST	AVL	ChainHashTable	HashTrieMap
insert	2404371.863	398262.1211	191343.8824	298906.8474
find	231054.9941	21292.1	5774.652941	17587.97059

```
public class GeneralPurposeDictionaryExperiment {
   private static int NUM_TESTS = 20000;
   private static int NUM_WARMUP = 3000;

   public static void main(String[] args) throws IOException {

       List<AlphabeticString> lst = new ArrayList<>();

       Scanner scn = new Scanner(new File("corpus/alice.txt"));
       while (scn.hasNext()) {
           AlphabeticString str = new AlphabeticString(scn.next());
           lst.add(str);
       }

       double totalTime = 0;
       for (int i = 0; i < NUM_TESTS; i++) {
           Random r = new Random(i);

           // Types of Dictionaries
           Dictionary<AlphabeticString, Integer> d = new
BinarySearchTree<>();
           Dictionary<AlphabeticString, Integer> d = new AVLTree<>();
           Dictionary<AlphabeticString, Integer> d = new
ChainingHashTable<>(MoveToFrontList::new);
           Dictionary<AlphabeticString, Integer> d = new
HashTrieMap<>(AlphabeticString.class);

           // Unsorted
           long startTime = System.nanoTime();
           for (AlphabeticString str : lst) {
               if (d.find(str) == null) {
                   d.insert(str, 1);
               } else {
                   d.insert(str, d.find(str) + 1);
               }
           }

           long endTime = System.nanoTime();
```

```
            // Sorted
            PriorityQueue<AlphabeticString> pr = new
PriorityQueue<>((a, b) -> a.compareTo(b));
            for (AlphabeticString str : lst) {
                pr.add(str);
            }

            //long startTime = System.nanoTime();
            while (!pr.isEmpty()) {
                AlphabeticString str = pr.poll();
                if (d.find(str) == null) {
                    d.insert(str, 1);
                } else {
                    d.insert(str, d.find(str) + 1);
                }
            }
            //long endTime = System.nanoTime();


            // Find
            int start = r.nextInt(lst.size() - 100);
            //long startTime = System.nanoTime();

            for (int j = 0; j < 100; j++) {
                d.find(lst.get(start + j));
            }

            //long endTime = System.nanoTime();
            if (NUM_WARMUP <= i) {
                totalTime += endTime - startTime;
            }

        }
        double averageRuntime = totalTime / (NUM_TESTS - NUM_WARMUP);
        System.out.println(averageRuntime);
    }

}
```

(6) uMessage

Use uMessage to test out your implementations. Using N = 3, uMessage should take less than a minute to load using your best algorithms and data structures on a reasonable machine.

How are the suggestions uMessage gives with the default corpus?
(here we mean spoken.corpus or irc.corpus, not eggs.txt)

Spoken.corpus: [the] [and] [of] [to]
I don't think these suggestions uMessage gives for spoke.corpus are pretty useful because none of the four suggested words has a noun that can be the subject first, or a personal pronoun like "I" or "You." It's not the way how we use to speak or write message. So, it's not effective for me to use the message to create a sentenc with comprehensive meaning.

Now, switch uMessage to use a cnorpus of YOUR OWN text. To do this, you will need a corpus. You can use anything you like (Facebook, Google talk, e-mails, etc.). We provide instructions and a script to format Facebook data correctly as we expect it will be the most common choice. If you are having problems getting data, please come to office hours and ask for help. Alternatively, you can concatenate a bunch of English papers you've written together to get a corpus of your writing.

PLEASE DO NOT INCLUDE "me.txt" IN YOUR REPOSITORY. WE DO NOT WANT YOUR PRIVATE CONVERSATIONS.

1. Follow these instructions to get your Facebook data. You will need to download it in JSON format. We also recommend onlys selecting your messages and limiting the date range to the past year to make the process go quicker.
2. Run the ParseFBMessages program in the p2.wordsuggestor package.
3. Use the output file "me.txt" as the corpus for uMessage. **Please make sure you do not commit your Facebook messages onto GitLab!**

How are the suggestions uMessage gives with the new corpus?

Me.txt: [i] [lol] [also] [doing]

The suggestions uMessages gives with me.txt is very useful. Since the file it used is the messages of chat in my Facebook. So, the words suggested catches the style of my language use. They are very common for me to use in the chat, like “i”, “lol.” So, this uMessage gives me four personality-suggested words.