Experimenting B+ Tree with Different Queries

Esra Kantarcı

1 What is B+Trees?

In order to get faster file systems, we need to provide better solutions for file management and accessing methods. For this purpose, we had seen indexing, sorting and many other operations. We had seen balanced binary trees are better approach for the searching purposes, but we can enhance the binary trees, too.

So, B+trees are somehow like balanced binary tree structures, but not in "bi"(2) sense: we use "m"-ary trees.

B+ Trees are self balancing trees with O(logn) complexity on each operation. B+ Trees' height is lower than binary trees, therefore the number of disk accesses are lower due to storing maximum possible keys in each B-Tree nodes.

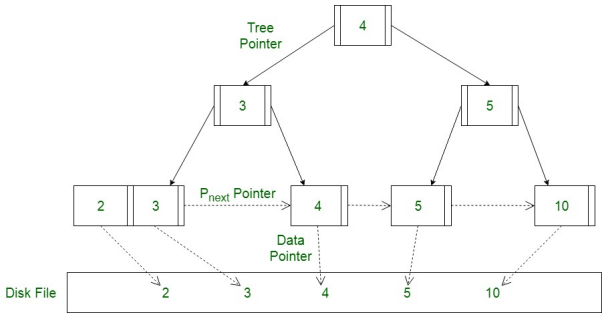

Figure 1: B+ Tree Structure

As you can see from the figure above, only leaf nodes have the data pointers to retrieve the data. This is B+Trees' one of the most significant difference from the B-Trees. Therefore, we have faster access at search methods, as we are going to experiment in the assignment.

2 Experiment Details and Figures

As I had been hospitalized 2 times in this week and got horrible times with my chin-bones and its neurotic issues, I had pretty limited time to prepare the experiment. The codes might have been more efficient, but could not improve much.

The https://github.com/shandysulen/B-Plus-Tree is used and in addition to its main class, I had created generate.java for the insertion of 500.000 randomized key-value pairs and ascending ordered key-value pairs, values are varied from -1000 to 1000. I added the randomized search queries with 100 search.

For the experiment purposes, I had chosen m as powers of 2: 2,4,8,16,32,64,128,256. As for the other task from the assignment, I had modified the switch-case of bplustree.java's "Search" part for keeping the track of time. After checking out the results, I went for microseconds for time unit, because I saw that, in average it is 0 milliseconds due to efficiency of the algorithm.

Since the hardware may affect the results of the experiment, so here is my specs:

Öğe	Değer
İşletim Sistemi Adı	Microsoft Windows 10 Pro
Sürüm	10.0.19042 Yapı 19042
Diğer İşletim Sistemi Açıklaması	Kullanılamıyor
İşletim Sistemi Üreticisi	Microsoft Corporation
Sistem Adı	DESKTOP-RPT65I4
Sistem Üreticisi	System manufacturer
Sistem Modeli	System Product Name
Sistem Türü	x64-based PC
Sistem SKU'su	SKU
İşlemci	AMD Ryzen 7 2700X Eight-Core Processor, 3700 Mhz, 8 Çekirdek, 16 Mantıks.
BIOS Sürümü/Tarihi	American Megatrends Inc. 4012, 14.06.2018
SMBIOS Sürümü	3.1
Ekli Denetleyici Sürümü	255.255
BIOS Modu	UEFI
Temel Kart Üreticisi	ASUSTeK COMPUTER INC.
Temel Kart Ürünü	PRIME X370-PRO
Temel Kart Sürümü	Rev X.0x
Platform Rolü	Masaüstü
Güvenli Önyükleme Durumu	Açık
PCR7 Yapılandırması	Bağlama Olanağı Yok
Windows Dizini	C:\Windows
Sistem Dizini	C:\Windows\system32
Önyükleme Aygıtı	\Device\HarddiskVolume1
Yerel Ayar	Türkiye
Donanım Soyutlama Katmanı	Sürüm = "10.0.19041.488"
Yüklü Fiziksel Bellek (RAM)	8,00 GB
Toplam Fiziksel Bellek	7,93 GB
Kullanılabilir Fiziksel Bellek	3,54 GB
Toplam Sanal Bellek	15,9 GB
Kullanılabilir Sanal Bellek	9,93 GB
Disk Belleği Dosyası Alanı	7,93 GB

Figure 2: Experimented PC's Properties

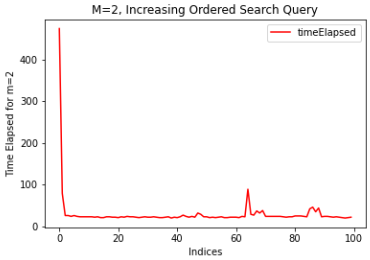

Figure 3: M=2 Ordered Values Inserted

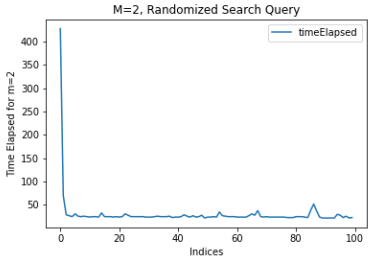

Figure 4: M=2 Randomized Values Inserted

The results were not so confusing, but pretty expected from an efficient algorithm. After trying out with different queries and inputs, here are the same search query with same inserted key-value pairs. The inserted values will also be attached to the results below for inspecting purposes.

As you can see, at the first search, it took higher than 400 microseconds, but then, the value was between 19 and 28 microseconds and there are several peaks and in the ordered inserted values, it is more deviated than the randomized valued key-value sets. Randomized valued insertions had more smooth graph.

When we check the query for the peak from key - ascending value B+ tree, the time was consumed going from 408655th to 45858th, and it took 89 microseconds.

As for the peak value from the randomized inserted values, the time was consumed going 262757th to 142495th, and it took 52 microseconds.

	Ordered	Randomized
2	29.89	30.28
4	28.06	29.94
8	29.07	28.34
16	29.43	30.85
32	35.92	36.53
64	37.04	38.94
128	36.93	39.39
256	36.53	45.21

Figure 5: Mean of the Time Elapsed of the Different M values for Each Query

Here are the other comparisons with different m values side to side:

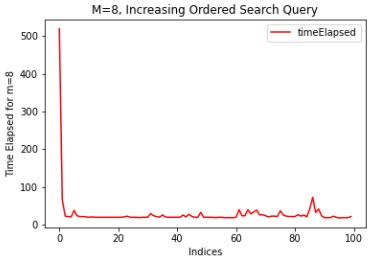

Figure 6: M=4 Ordered Values Inserted

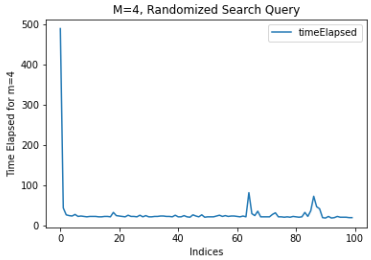

Figure 7: M=4 Randomized Values Inserted

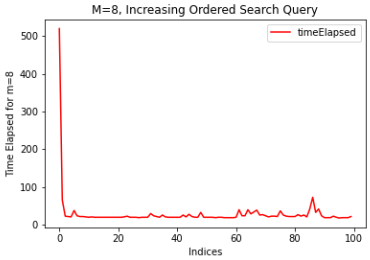

Figure 8: M=8 Ordered Values Inserted

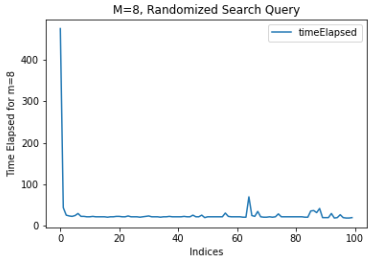

Figure 9: M=8 Randomized Values Inserted

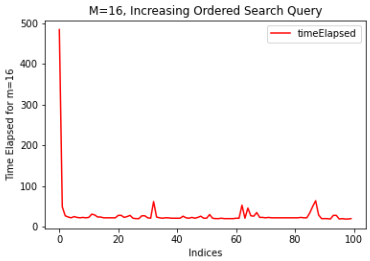

Figure 10: M=16 Ordered Values Inserted

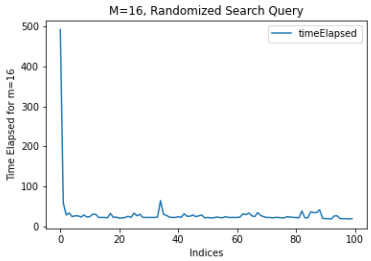

Figure 11: M=16 Randomized Values Inserted

3

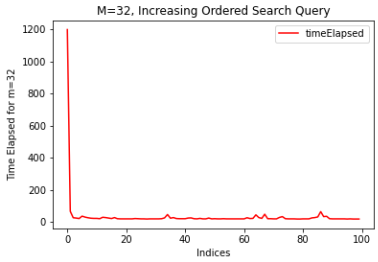

Figure 12: M=32 Ordered Values Inserted

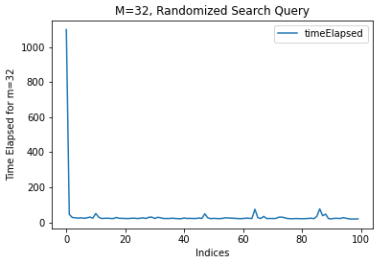

Figure 13: M=32 Randomized Values Inserted

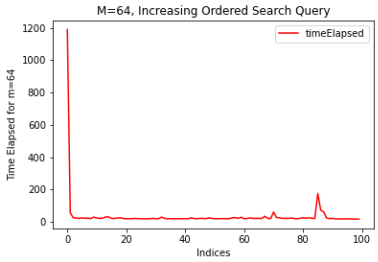

Figure 14: M=64 Ordered Values Inserted

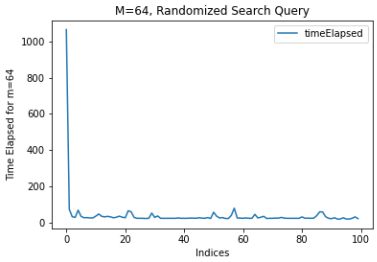

Figure 15: M=64 Randomized Values Inserted

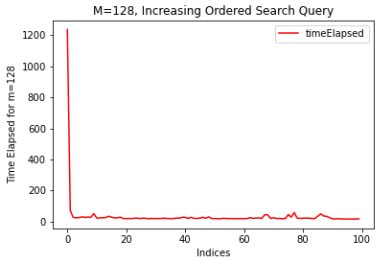

Figure 16: M=128 Ordered Values Inserted

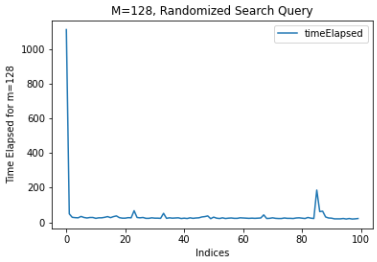

Figure 17: M=128 Randomized Values Inserted

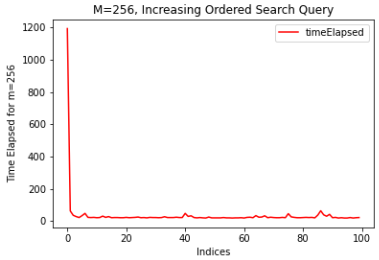

Figure 18: M=256 Ordered Values Inserted

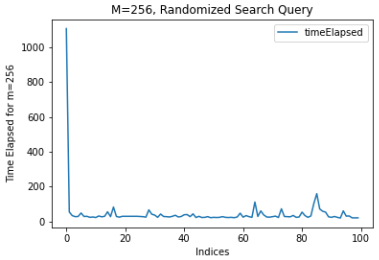

Figure 19: M=256 Randomized Values Inserted

4 A Summary of Comparisons

As you can see, in the randomized inserted values, the average time spent per search query in microseconds were slightly higher than the orderly generated and inserted values. And there is a (kind of) drastic change in the average time between m=16 and m=32 trees. But then again, this is because of the first operation's time getting increased at every time the m value gets bigger.

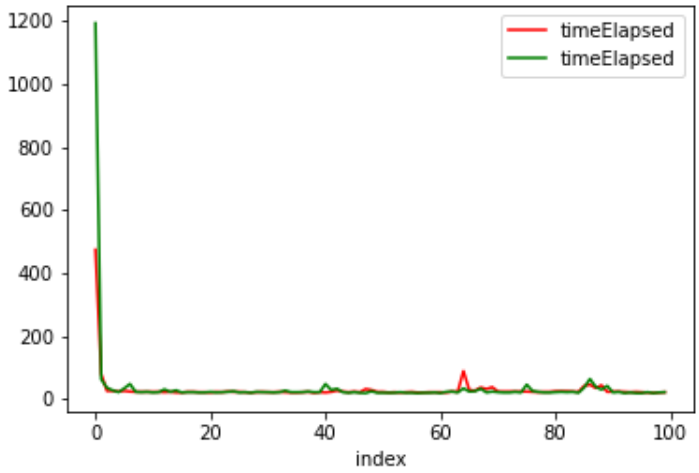

Figure 20: Ordered 2 (in Red) vs Ordered 256 (in Green)

	Ordered	Randomized
2	2989	3028
4	2806	2994
8	2907	2834
16	2943	3085
32	3592	3653
64	3704	3894
128	3693	3939
256	3653	4521

Figure 21: Sum of Time Taken by Search Queries on B+Trees at Different M values (in microseconds)

	Ordered	Randomized
2	29.89	30.28
4	28.06	29.94
8	29.07	28.34
16	29.43	30.85
32	35.92	36.53
64	37.04	38.94
128	36.93	39.39
256	36.53	45.21

Figure 22: Average Time of Search Queries on B+Trees at Different M values (in microseconds)

5 Conclusion

In conclusion, B+ trees are such efficient and strong algorithms that even inserting 500.000 key-values then searching 100 random keys are really fast that we needed to measure it with microseconds.

The time consuming one was always the first one, regardless the inserted values' context (even if it is randomized or in ascending order). The other peaks were due to re-setting the pointer to a point which is relatively far away from the current position, as it was expected.

The randomized values' graphs were smoother than the ascending ordered inserted values, this was also expected from the nature of the pointers: when the distribution is more uniform(which is also the main point of hashing), the access time is similar to each (because we were using randomized search queries as well.)

In short, B+ Trees are widely used in DBMS and other file structures and fast access with efficient storage is the reason.

6 Code of generator.java

```
import java.io.*;
import java.util.*;

public class generator {

    public static void main(String[] args) {
        // Create an empty hash map
        HashMap<Integer, Double> map = new HashMap<>();
        caseRandom(map);


        HashMap<Integer, Double> mapOrdered = new HashMap<>();
        caseIncreasing(mapOrdered);



        // Print size and content
        System.out.println("Size of map is: "
                + map.size());

                String[] query = new String[100];

        for(int i = 0; i<100; i++) {
             query[i]= randomSearchQuery();
        }

        try (Writer writer = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream("inputgeneratedRandom.txt"), "utf-8"))) {


            ((BufferedWriter) writer).newLine();

            Iterator iterator = map.entrySet().iterator();
            while (iterator.hasNext()) {
                Map.Entry me = (Map.Entry) iterator.next();
                String insertOrder = inserter((int)me.getKey(), (Double)me.getValue());
                writer.write(insertOrder);
                ((BufferedWriter) writer).newLine();

            }
            for(int i = 0; i<100; i++){
               String search = query[i];
               writer.write(search);
               ((BufferedWriter) writer).newLine();
```

```
        }

        writer.close();

    } catch (IOException e) {
        e.printStackTrace();
    }

    try (Writer writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream("inputgeneratedOrdered.txt"), "utf-8"))) {


        ((BufferedWriter) writer).newLine();

        Iterator iterator = mapOrdered.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry me = (Map.Entry) iterator.next();
            String insertOrder = inserter((int)me.getKey(), (Double)me.getValue());
            writer.write(insertOrder);
            ((BufferedWriter) writer).newLine();

        }
        for(int i = 0; i<100; i++){
            String search = query[i];
            writer.write(search);
            ((BufferedWriter) writer).newLine();
        }

        writer.close();

    } catch (IOException e) {
        e.printStackTrace();
    }

}



public static void caseRandom(HashMap<Integer, Double> map) {
    // Add elements to the map
    for (int i = 0; i < 500000; i++) {
        Random rand = new Random();
        double randomValue = -1000 + (2000) * rand.nextDouble();
        map.put(i, randomValue);
    }
```

```
    }

    public static void caseIncreasing(HashMap<Integer, Double> map) {
        // Add elements to the map
        int j=0;
        double[] randomNumbers = new double[500000];
        double delta = 2000.0 / (float)500000;
        Random random = new Random();
        for (int i = 0; i < randomNumbers.length; i++) {
            randomNumbers[i] = (double)((i*delta + random.nextDouble() * delta)-1000);
        }
        for (int i = 0; i < 500000; i++) {
            map.put(i, randomNumbers[j]);
            j++;

        }
    }

    public static String randomSearchQuery(){
            Random rand = new Random();
            int key = rand.nextInt(500000);
            return "Search (" + key + ")"  ;

    }

    public static String initializer(int m) {
        return "Initialize(" + m +")" ;

    }

    public static String inserter(int key, Double value){
        return "Insert("+key+", "+value+")";
    }


}
```

References

[1] Github - ShandySulen - B Plus Tree
`https://github.com/shandysulen/B-Plus-Tree`

[2] GeeksForGeeks - B+Trees
`https://www.geeksforgeeks.org/introduction-of-b-tree-2/`