AKDENIZ UNIVERSITY

COMPUTER ENGINEERING

CSE 443

FILE ORGANIZATION AND PROCESSING

HW01 – B+ TREE PROJECT

by

Tuğba Güler

Table of Contents

1. B + Trees

The B + tree works very quickly and efficiently when accessing, adding, or deleting the data in sorted form. It uses the indexing method while working. Indexing is done by selecting a value in each data and specific to that data only. This selected value is the key value.

B + trees have a dynamic structure. It can grow and shrink depending on the number of data or keys it has. In B + trees, all leaves are the same height and all records are kept at leaf level. Only keys are held at non-leaf (internal node) points. In the B + tree, leaf nodes are linked using a link list. Therefore, a B + tree can support random access as well as sequential access.

B + trees contain internal nodes and leaf nodes. In these trees, each leaf node is equidistant from the root node. The B + tree is at the n level where n is constant for each B + tree. An internal node of the B + tree contains at least n / 2 register pointers outside the root node and can contain at most n pointers. A leaf node contains at least n / 2 record pointers and n / 2 key values, while a leaf node contains at most n record pointers and n key values.

The B + Tree is used to store large amounts of data. Main memory is limited in size. The internal nodes (access keys to the records) of this percentage B + tree are stored in main memory and leaf nodes are stored in secondary memory.

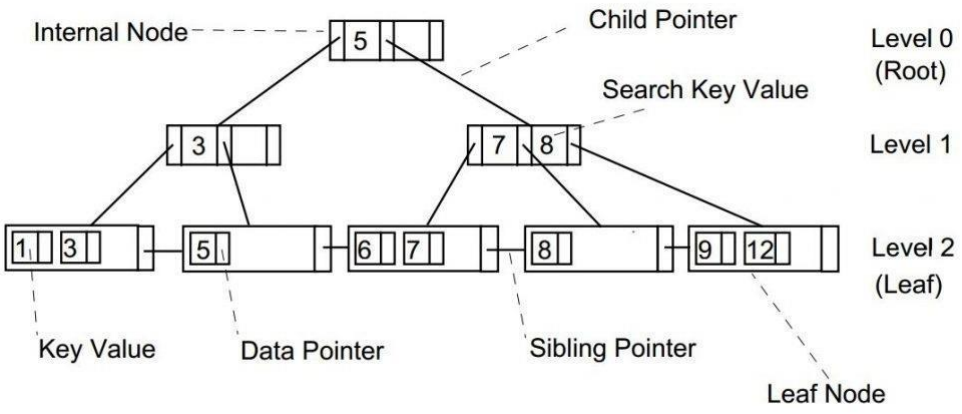


Figure 1.1 B + Tree Structure

As seen in Figure 1.1, all of the records in the tree are in order. All nodes at the leaf level where the records are kept are in order both within themselves and to each other. Being ordered relative to each other means that all elements inside one of the two nodes have a smaller key value than those inside the other. That is, the values chosen as keys should be compared with each other. Large and small cases (like numbers in ascending order) are used in numbers. Alphabetical order is taken into consideration in the articles. Keys are kept in sequence in the same way on all nodes. There are markers on the front and back of each of the switches that indicate the nodes of the next lower level. The bottom nodes are already in order, and the value of the key to the left of the pointer that points it is less than or equal to the value of the key with the minimum value in the underlying node. Also, the key value to the right of the marker is greater than any key value in the node marked. If there is no key value to the left or right of the node, the value to the right or left of the marker showing that the node is checked. The parental nodes lookup process is done by repeating itself for each key value that is not found. If a value cannot be reached at the end and the key viewed first is at the very beginning of the node, it indicates that the node has no lower bound. Similarly, if it is at the end of the node, it indicates that the node has no upper bound.

- **When searching for a node with key values;**

Step 1: The root node is selected as the node under view.

Step 2: If the node under view is not a leaf;

- The search is made according to the key value of the searched record in the node being looked at.
- Go to the node pointed to by the marker after the key with the highest value smaller than the search key value.
- This node is selected as the node under view.
- Return to step 2.

Step 3: If the node being viewed is a leaf, the node being searched is either on that leaf node or is not in the records. Therefore, the node is also searched. If the searched value exists in this node, the result is reached.

- **B + tree insertion algorithm**

Case 1: According to the key value of the record to be added, the node that should be is found.

Case 2: If this leaf node records less than twice its degree; that is, if there is space in the node, the record is inserted at the correct place within the node, keeping the order. (and here the algorithm ends.)

Case 3: If there is no space, the node is divided into two-part.

Case 4: A new leaf is created and half of the elements in the node are added to this new leaf without breaking the order.

Case 5: A new leaf is created and half of the elements in the node are added to this new leaf without breaking the order.

Case 6: If the parent node is full, it splits in two-part.

Case 7: Then the switch in the middle of it is just shifted up, in other words, it passes to its parent, and unlike leaf nodes, its duplicate is not left below.

Case 8: This process is repeated until a parent node that does not require splitting is found.

Case 9: If the last root node is also divided into two, a new root node with a single key-value and two markers are created.

- **B + tree algorithm used to delete a record with a key**

Case 1: The record with the key is found among leaf nodes. If the records kept in the node are equal to or greater than the grade of the tree, deletion is performed.

Case 2: If there are fewer records, that is, they are borrowed from the next node with the same parent and added to this node.

Case3: If this operation fails, that is, if the sibling node is also registered below the rank, the two nodes are merged.

Case 4: If merged, the marker showing the node on the right and its key are deleted from the parent. In this case, while the number of keys decreases, the parent is also subjected to a deletion operation up to the root node.

2. Experimenting the B + Tree Implementation

```
package hw01_file_bplustree;


import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
/**
 *
 * @author Tuğba Güler
 * student number : 20150807012
 *
 */
public class HW01_File_BPlusTree {
   public static void main(String[] args) throws IOException {
       long start = System.nanoTime();
       //The writing process is handled in this section.
       FileWriter writting_to_input_file = new FileWriter("C:/input.txt");


       BufferedWriter out = new BufferedWriter(writting_to_input_file);


       // Writes each of its command-line arguments on a line by itself
       for (String tgb : input_file_for_bpluss_tree()) {
           out.write(tgb + "\n");
       }
       out.close();


       // Calculated how long the program was running
       long finish = System.nanoTime();
       System.out.println("Working time of the entire program : " + (double)(finish-start)/1000000);
   }
```

```
  public static List<Double> generate_numbers() {
      ArrayList<Double> list = new ArrayList<Double>();
      //key-value pairs must be randomly generated only once for 500000
      int number_of_generate_numbers = 500000;
      for (int i = 0; i < number_of_generate_numbers; i++) {
          // The values must be floating point numbers in the range -1000.0 to +1000.0
          // Java has two primitive types for floating-point numbers:float and double
          // Codes organized using double
          double x = (double) (Math.round(((Math.random() * ((2000) + 1)) - 1000) * 10000.0) / 10000.0);
          // added to the list
          list.add(x);
      }
      // list sorted in ascending mode
      Collections.sort(list);
      return list;
  }
  private static Iterable<String> input_file_for_bpluss_tree() {
      /*----- m value is created -----*/
      long start_created_m_value = System.nanoTime();
      // A list of commands has been created
      List<String> command_list = new ArrayList<String>();
      // m values created between 5 and 50
      int m = (int) (((50 - 5) * Math.random()) + 5);
      // Command to start printing
      command_list.add("Start --> " + m );
      System.out.println("m : " + m);
      long finish_created_m_value = System.nanoTime();
      System.out.println("Calculating creating different m value time : " + (double)(finish_created_m_value-
start_created_m_value)/1000000);

      /*-----  Insert queries created----- */
      // A hash map of the key value has been created
      // Maps can take any Object, for both key and value.
      long start_inset_numbers = System.nanoTime();

      Map<Integer, Double> map = new HashMap<Integer, Double>();
```

```
        // Keys will be generated and these keys will be associated with the generated values.
        int t = 1;
        for (Double g : generate_numbers()) {
            map.put(t, g);
            t++;
        }
        // Insert commands have been printed
        for (Integer c : map.keySet()) {
            command_list.add("Insert(" + c + "," + map.get(c) + ")");
        }
        long finish_inset_numbers = System.nanoTime();
         System.out.println("Calculating inset numbers : " + (double)(finish_inset_numbers-start_inset_numbers)/1000000);


        /*----- Search queries created -----*/
        // To calculate how long the search process takes
        long start_searching_time = System.nanoTime();
        // 100 unique numbers generated for keys
        ArrayList<Integer> keys = new ArrayList<Integer>();
        int number_of_generate_keys = 100;
        for (int j = 1; j <= number_of_generate_keys; j++) {
            int n = (int) ((Math.random() * ((100) + 1)) - 1);
            // added to the list
            keys.add(n);
        }
        // Search queries added for each key
        for (int k : keys) {
            command_list.add("Search(" + k + ")");
        }
         long finish_searching_time = System.nanoTime();
         System.out.println("Calculating searching time : " + (double)(finish_searching_time-start_searching_time)/1000000);
        // return list of commands
        return command_list;
    }
}
```

Code Output:

m : 13

Calculating creating different m value time : 0.6818

Calculating inset numbers : 527.8055

Calculating searching time : 0.2337

Working time of the entire program : 659.7078

3. Test Results

m values	Time spent for choosing different m values	Time spent for total search time	Time Spent for Insert Operation	Working time of the entire program
6	0.9345	0.251	516.6734	619.5205
7	0.9723	0.2358	620.1883	768.8901
8	0.5547	0.2504	543.0523	630.1151
11	0.9655	0.1848	569.7682	687.2336
12	0.7914	0.2486	489.9454	589.6036
14	0.9813	0.1952	651.3780	796.4565
16	0.5570	0.1675	509.9015	625.854
17	0.8698	0.2009	498.9710	600.8325
18	0.6188	0.2278	510.0642	611.0516
20	0.6112	0.2247	539.2421	636.7557
22	0.5825	0.2175	526.9450	631.3496
23	0.5766	0.2510	546.3801	647.3895
23	0.8266	0.1791	544.9348	647.0956
24	0.8688	0.2278	589.7102	695.9211
25	0.8849	0.2023	523.0715	638.8520
26	0.6390	0.1419	512.1778	632.9385
27	0.5518	0.3076	512.6054	626.3487
28	0.6886	0.2084	532.4098	632.4360
31	0.9183	0.2249	659.7172	785.6867
32	0.6797	0.1764	661.6930	793.3476
33	0.8711	0.2889	505.7151	605.9135
34	0.5790	0.2334	533.9906	611.7944
34	0.7381	0.1822	528.8904	652.9178
36	0.7735	0.1720	542.6207	657.1616
38	0.8607	0.3371	511.0463	612.7457
41	0.6327	0.2401	500.9801	610.2074
42	0.6391	0.2036	507.4862	618.5271
43	0.6008	0.2406	509.6785	616.8800
48	0.8201	0.2600	645.9042	781.7992
49	1.0124	0.2692	543.6891	653.6535

Table 3.1 Time comparison output of the code

The codes written above have been run many times and 30 of the outputs obtained as a result of the execution of the code are shown in this table.

The m values were randomly generated between 5 and 50. Therefore, the same m value can be recalculated during the test. Since the values are given randomly, each time the code is run, it gives different results. This situation is shown in the table for m values 23 and 34.

Times given are calculated in nanoseconds. One nanosecond is 1.0×10^{-9} seconds. This means that the code written works very fast. If we specify the running speed of the program in seconds when the value of m is selected as 6;

- Selection time of m value: 9.345×10^{-10} s
- Time elapsed during the search process: 2.51×10^{-10} s
- Time elapsed during the piecing operation: $5{,}1667 \times 10^{-7}$ s
- Time taken for the entire program to run: $6{,}1952 \times 10^{-7}$ s

For the table to be interpreted more easily, in other words, the graphics created using this table are shown below. 10 different m values were selected from this table and these values and their results are shown in the tables.

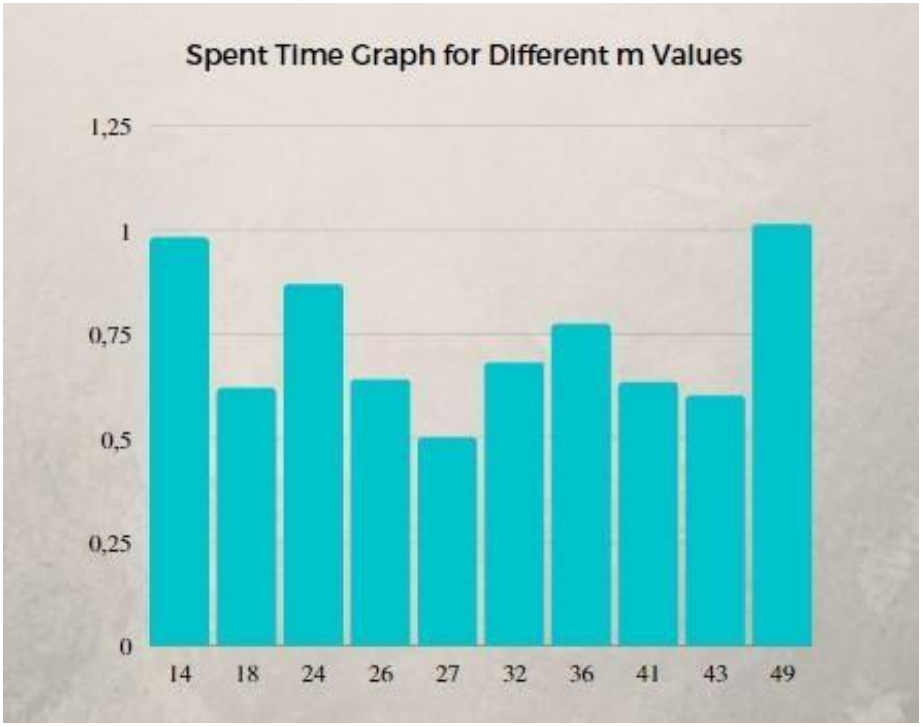


Table 3.2 Spent Time for Different m Values

The numbers 14, 18, 24, 26, 27, 32, 36, 41, 43, and 49 indicated in the horizontal row are randomly selected m values. The times to determine these values are shown in the graphic. Generally, run times are between 0.5 and 0.75 nanoseconds.

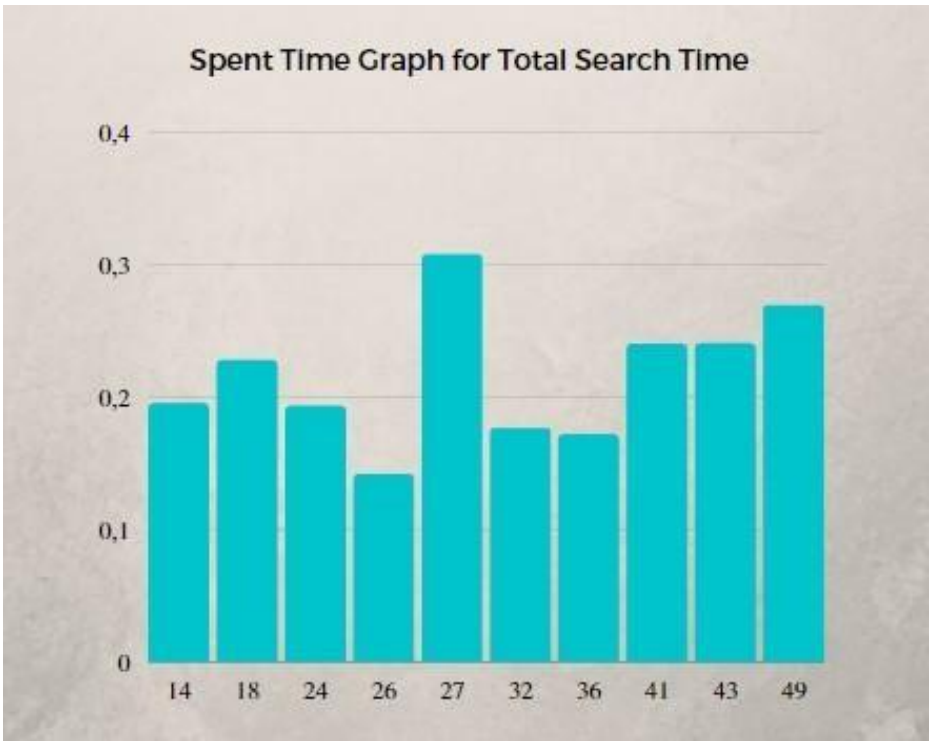


Table 3.3 Spent Time for Total Search Time

The times for 100 search operations performed according to the selected m values are shown in Table 3.3. In general, it is not seen that the search process takes place in a very short time. Elapsed time frames are between 0.1 and 0.3 nanoseconds.

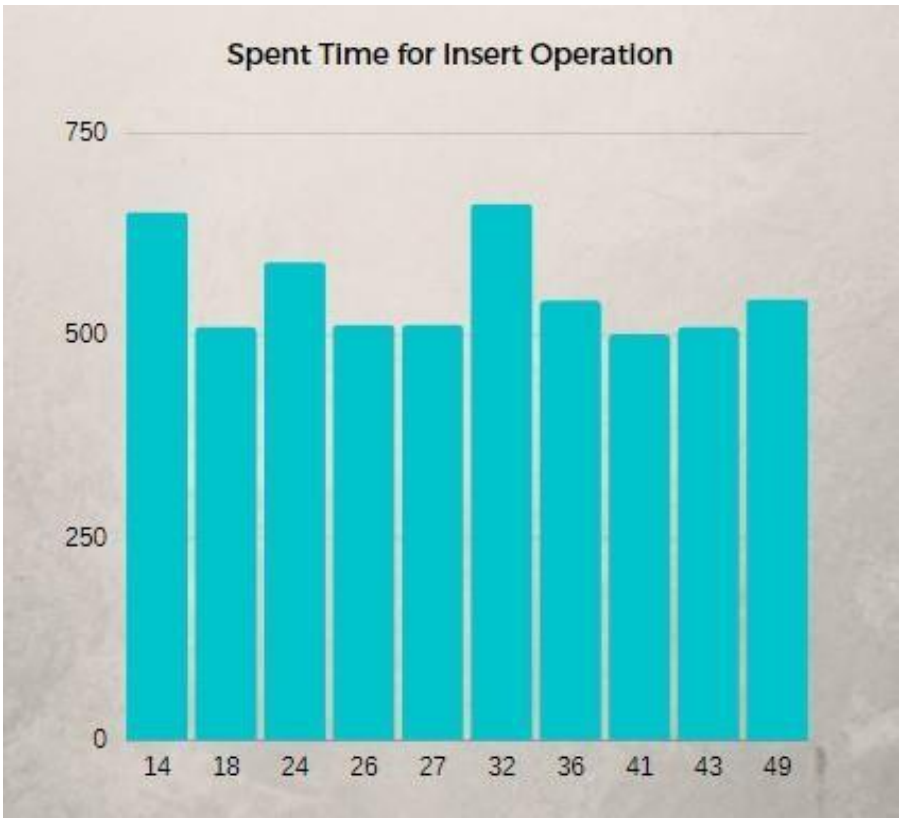


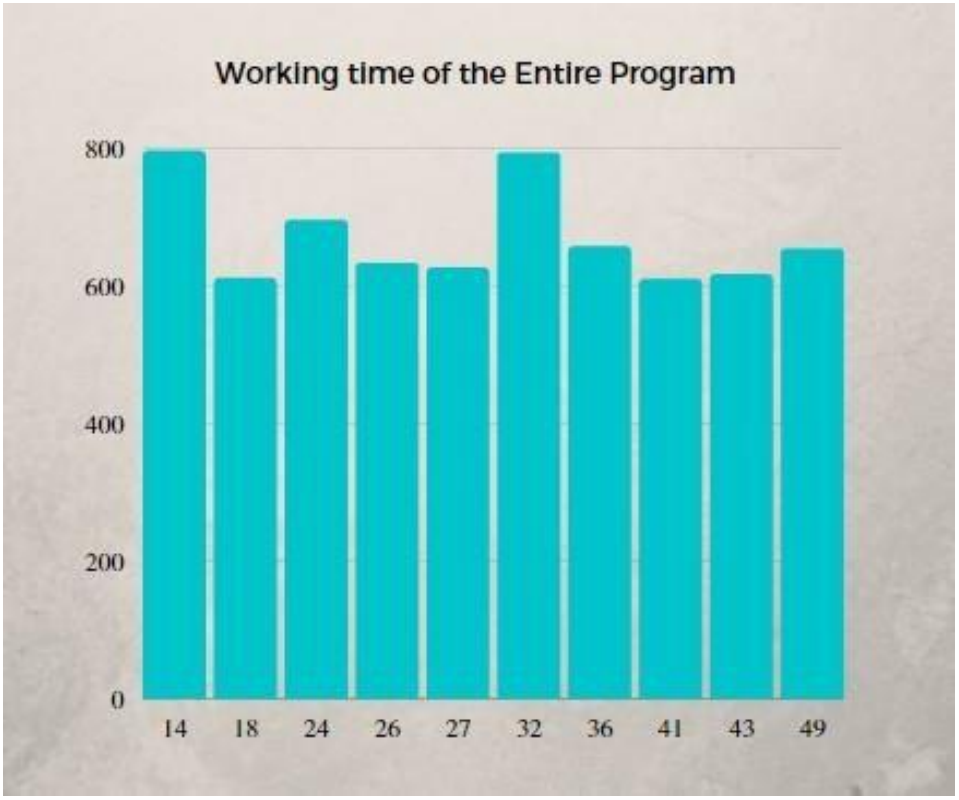
Table 3.4 Spent Time for Insert Operation

The table 3.4 shows the time elapsed during the insertion operations. In the code, this addition is made 500000 times. Therefore, the addition process takes much longer than the determination of the m value and the search operations. As can be seen from the table, the elapsed time is between 500 and 700.

When comparing search and insertion operations, it is seen that the search process runs much faster than the insertion process. To have a clearer evaluation, the number_of_generate_keys value in the field where the search queries are written in the code was changed to 500000 and the code was run again. Results obtained for m 38 value:

- Calculating creating different m value time: 0.805
- Calculating inset numbers: 536.9258
- Calculating searching time: 74.2685
- Working time of the entire program: 708.1214

As can be seen in these results, **the search process works much faster than the insert operation**.



3.5 Working Time of the Entire Program

The time spent taken to run the program is high. During the time it takes for the program to run, there is a creation of the m value, the realization of the insert operation, the search operations, and the printing of these results to the input.txt file. The insert process has been made 500000 times and is a very time-consuming process. However, it should not be forgotten that the obtained value results are in nanoseconds. So, the speed of operation is much higher than many algorithms. According to the results obtained from the codes written, most of the time spent running the program is between 600 and 700. In some cases, it approached 800.

REFERENCES

1- http://sayef.tech/post/b-plus-tree-tutorial/
2- https://www.javatpoint.com/b-plus-tree
3- https://en.wikipedia.org/wiki/B%2B_tree
4- https://www.geeksforgeeks.org/introduction-of-b-tree/
5- https://e-bergi.com/y/b-agaclari/
6- https://medium.com/analytics-vidhya/multi-way-search-trees-c193e2b36998