Selection of the Appropriate Database Index for Data Storage

Research Question: To what extent does the structure of a B-tree make it a better choice than Extendible Hash Tables for creating database indexes?

By: Pratik Choudhary

International School Manila

IB Candidate Number: 000046-0026

Examination Session: May 2015

IB Computer Science Extended Essay

Advisor: Mr. Justin Robertson

EE Body Word Count: 3762

Name: Pratik Choudhary IB Candidate Number: 000046-0026

Abstract

The aim of the paper is to answer the research question: *To what extent does the structure of a B-tree make it a better choice than Extendible Hash Tables for creating database indexes?* The investigation to the question consists of two parts. The first part of the investigation aims to understand both the abstract data structures, B-trees and Extendible Hash Tables. To improve the understanding of the structure of these indexes, these indexes are coded in Java and tested thoroughly. By running multiple tests on the code, understanding of insertion, deletion and search methods is deepened. The second part of the investigation aims to compare the databases indexes by analyzing their strengths and weaknesses in terms of disk access. The investigation also consists of minor mathematical derivations and calculations to understand how the structure of each index results to the number of disk accesses that occurs in both exact and range queries. Then, the database indexes are compared under the context of an actual database, namely the Oxford English Corpus, which consists of roughly 2 Billion words and stores many English files and texts within it.

The purpose of this research paper is to help solve the dilemma of programmers and database designers as they choose the appropriate database index for their database. The findings of the paper suggests the Extendible Hash Tables are the most powerful database index for performing exact queries while B-trees are better in performing range queries. However, due to the versatility of B-trees, their average efficiency in both the queries is better than Extendible Hash Tables. Hence, databases that perform range queries or both should use B-trees while those databases that only perform exact queries should use Extendible Hash Tables.

Word Count: 285

Table of Contents

1 Introduction

Storing data is an important task done by all ages. Everyone has some sort of data or information that they want to store in some form. A toddler stores his drawings, a teenager stores his homework from school, a businessperson stores his year-round financial records, and a retired employee stores his bank transactions and insurance details. Everyone needs to store data in one form or another. Anything they want to store is data or information.

To store data, some prefer using hardcopies such as paper. The teenager would most likely store his math homework in his math notebook or a three-ring binder. The businessperson or the retired employee may not opt for storing their vital information in hardcopy. Many choose to store their valuable data and information in database systems.

A Database system is “a collection of information organized in such a way that a computer program can quickly select desired pieces of data” (Beal). Databases store fields, records and files. A field is a single piece of information, a record is a set of fields and a file is a group of records (Beal). Databases store data in tabular form, mainly in the secondary memory. The size of a database can range from a byte to thousands of terabytes. The question is how can the system possibly access a particular field or record out of billions stored in memory in a short amount of time? The answer to that is very simple. Database indexing. Through database indexing, the system can extract data from a gigantic database in a couple of seconds.

What is a database index? A database index is like the table of contents in a book. Each row of the table of contents consist of a string that represents a section of the book and a page number that represents the address of the section in the book. Database indexing follows a similar concept. It consists of a list of data field, each representing a record in the database. This paper will refer to the data field that represents a record as an ID. Parallel to the ID of each record is its address in the secondary memory. When searching for a record in the database, the system searches for the ID that represents the record in the index to get access to the address in the secondary memory that leads the system to the record.

There are basic indexes such as arrays and linked lists. These indexes are the most basic data structures used for indexing. The system is required to move sequentially through all the IDs when searching for an element. An important factor that makes an index powerful is its structure because the structure can reduce the number of IDs that the system would have to read which would eventually save time and hence be fast. Only a fast index is a powerful index.

Two highly powerful known abstract data structures used as database indexes are B-trees and Extendible Hash Tables. These indexes are alternatives to each other. When creating a database, a common dilemma faced by programmers is to choose the database index that best suits the requirements of the organization. This research paper will address this dilemma by asking the question:

To what extent does the structure of a B-tree make it a better choice than Extendible Hash Tables for creating database indexes?

This research question is important to investigate because it will work as a guide for future organizations when selecting the appropriate database index to support their database and provide the best performance.

B-tree and Extendible Hash tables are two popular dynamic indexes. These two indexes contrast greatly in terms of structure. Each index has its own strengths and weaknesses. The index that has the best performance overall amongst all the tasks it is provided will be the one that best suits the requirement of the organization.

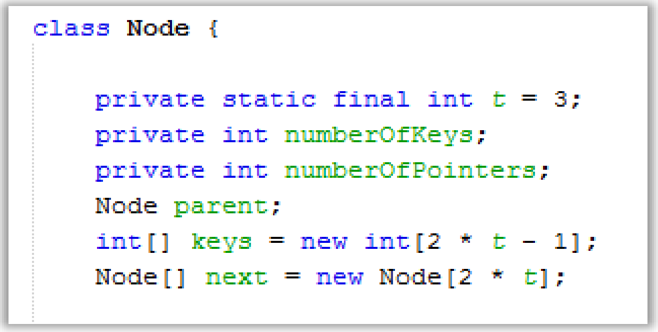
This paper will investigate the different types of tasks or queries that either of the indexes are required to perform and will decide which database index is the better one. The answer to this investigation will provide help for those programmers who do not have any knowledge of the possible requirements of their database and simply want to choose the best possible index they can get. This paper investigates an actual database that uses both the indexes mentioned above. The Oxford English Dictionary Corpus "often switch to a different data structure to support indexing, for example, B-trees or external hash tables" (Healey, File Indexing, 27). Instead of looking at the external hash tables, this paper will look at extendible hash tables, which are very similar to external hash tables.

The Oxford English Dictionary Corpus is a database consisting of more than 2 billion words. "The database consists of 20 different major subject areas, including categories such as society, medicine, leisure, and the startling large segment, news, from which 24.4 percent of the database's constituent words come. Subgroupings are then organized in each of these major categories" (Guz). Even now, the users add new words and texts created in English into the corpus. This paper will discuss which index is more suitable for this enormous database.

The hypothesis is that overall B-trees are the better choice for database indexing compared to Extendible hashing because of the versatile structure of the index and the arrangement of nodes in the index. The hypothesis will argues that the arrangement of nodes and pointers provide an advantage over Extendible hashing when performing large-scale tasks.

2 B-Tree

B-trees are ubiquitous abstract data structures, also known as balanced trees, are used for database indexing. Each node of a b-tree represents a disk page (Zhang). Unlike its predecessor, the Binary tree, B-tree allows the nodes to store more than one ID and two pointers. The structure of the tree have three different layers, a root node, internal nodes, and leaf nodes. The root node is able to carry from one to $2t - 1$ IDs. The variable t is the class variable of a node, and it depends on the size of each disk block. This variable then decides the maximum number of elements that the node can carry.



```
class Node {

    private static final int t = 3;
    private int numberOfKeys;
    private int numberOfPointers;
    Node parent;
    int[] keys = new int[2 * t - 1];
    Node[] next = new Node[2 * t];
```

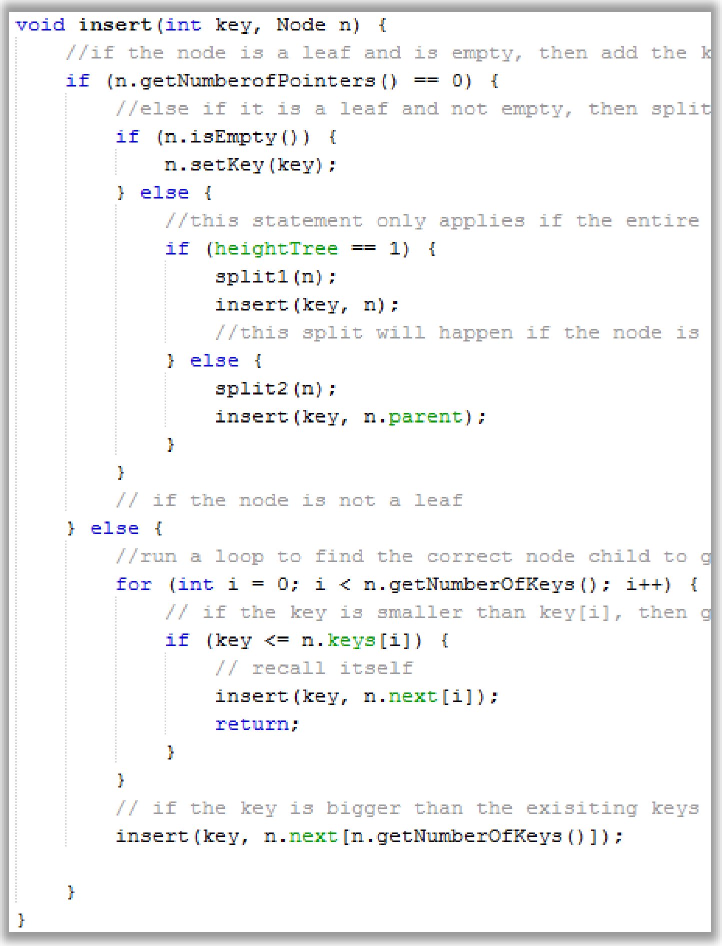
Figure 1: *B-tree Node class variables. Coded by Pratik Choudhary.*

The internal nodes and the leaf nodes carry from $t - 1$ to $2t - 1$ IDs. The root also points at two to $2t$ nodes while the internal nodes point at t to $2t$ nodes (Neubauer). The node class showed in **Figure 1** shows the variables that each node object carries. This node class consists of two parallel arrays, one for IDs or keys and the other for pointers or next. One variable that the code misses is the address of the record in the secondary memory.

Since B-trees are balanced trees, all the leaf nodes have the same height/depth in the tree. This means that the number of nodes that the system will go through in order to find the array is the same for all leaf nodes. Since each time the system reads a node, it has to access the secondary memory; therefore, the number of disk access it takes for the system to reach any leaf node is also same. This is an important factor in the structure when performing the insertion method.

2.1 Insertion

In a B-tree, the system inserts the IDs in a particular order depending on the ID type. If the IDs are integers, then they are stored in ascending order. In the code from **Figure 2**, the algorithm adds an ID into the tree. In B-trees, the system reads from the root, goes through the internal nodes, and finally reaches the leaf nodes. When insertion takes place, the system goes to the root, reads the ID to identify which pointer it wants to move into, and then determines the child node.



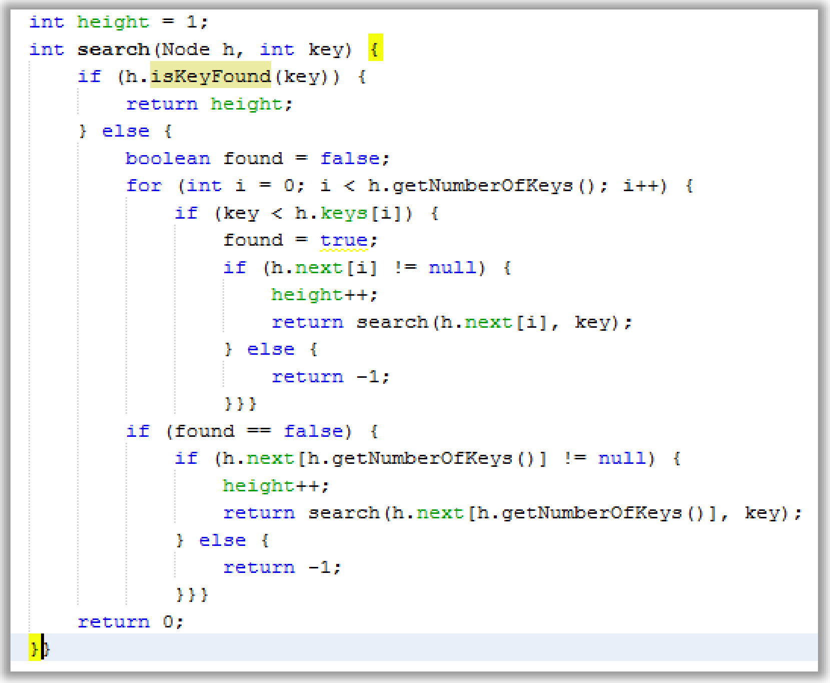
```
void insert(int key, Node n) {
    //if the node is a leaf and is empty, then add the k
    if (n.getNumberofPointers() == 0) {
        //else if it is a leaf and not empty, then split
        if (n.isEmpty()) {
            n.setKey(key);
        } else {
            //this statement only applies if the entire
            if (heightTree == 1) {
                split1(n);
                insert(key, n);
                //this split will happen if the node is
            } else {
                split2(n);
                insert(key, n.parent);
            }
        }
        // if the node is not a leaf
    } else {
        //run a loop to find the correct node child to g
        for (int i = 0; i < n.getNumberOfKeys(); i++) {
            // if the key is smaller than key[i], then g
            if (key <= n.keys[i]) {
                // recall itself
                insert(key, n.next[i]);
                return;
            }
        }
        // if the key is bigger than the exisiting keys
        insert(key, n.next[n.getNumberOfKeys()]);

    }
}
```

Figure 2: *Insertion Algorithm. Coded by Pratik Choudhary*

It continues this process recursively until it reaches the correct leaf node, where the system inserts the ID. If the leaf is full, an extensive process of node split runs and the system creates a new node that has half of the IDs of the original leaf node. The number of disk accesses that occurs in the process of insertion is equal to the height of the tree.

2.2 Search

In the code from **Figure 3**, the system starts searching from the root and moves towards the leaf node. Unlike the successor, B+tree, the B-tree stores data in the leaf nodes as well as the in the root node and internal nodes. Therefore, the B-tree recursively moves down the tree until it finds the ID and the address of the record.



```
int height = 1;
int search(Node h, int key) {
    if (h.isKeyFound(key)) {
        return height;
    } else {
        boolean found = false;
        for (int i = 0; i < h.getNumberOfKeys(); i++) {
            if (key < h.keys[i]) {
                found = true;
                if (h.next[i] != null) {
                    height++;
                    return search(h.next[i], key);
                } else {
                    return -1;
                }}}
        if (found == false) {
            if (h.next[h.getNumberOfKeys()] != null) {
                height++;
                return search(h.next[h.getNumberOfKeys()], key);
            } else {
                return -1;
            }}}
    return 0;
}}
```

Figure 3: Search Algorithm for B-tree Coded by Pratik Choudhary

The number of disk accesses is less than or equal to the height of the tree because it can find the ID in the leaf node or in the internal nodes. If found in the internal node, the number of disk access is less than the height of the tree, otherwise the number of disk access is equal to the height of the tree.

2.3 Deletion

Like the search method, the system searches for the ID that the user wants to delete. Once found, the system checks if the node is a leaf or an internal node. If it is a leaf and the number of elements in the node is $t-1$, the system merges the node with one of the neighbor nodes, otherwise the system deletes the ID from the keys array. If the node is not a leaf, the system swaps the ID with the median of the appropriate leaf node. Then it deletes the ID from the leaf node.

3 Extendible Hash Tables

Extendible hash tables, the alternatives of B-trees, are equally ubiquitous and powerful database indexes. As B-trees consist of nodes and pointers, Extendible hash tables consist of a directory and multiple buckets. A directory is in "the form of an array with 2^d entries, each entry storing a bucket address. The variable d is the global depth of the directory. To decide where a key k is stored, Extendible Hashing uses the last d bits of some adopted hashing function h(k) to choose the directory entry" (Zhang). The system passes the ID into the hashing function that then produces a set of bits that represent the ID. Once that is completed, it matches the last d bits of the ID to the appropriate directory entry with the same last d bits.

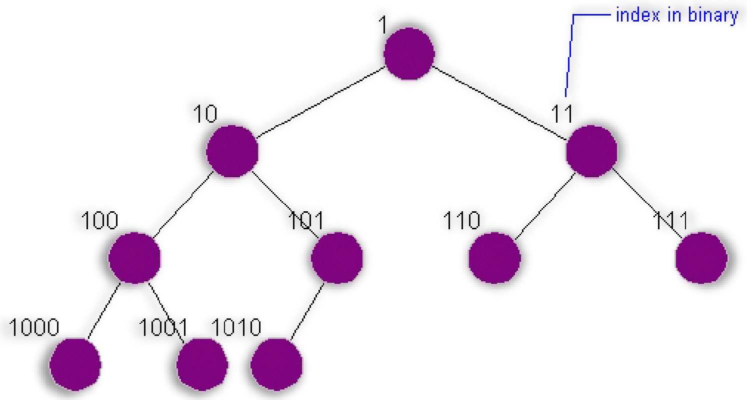


***Figure 4**: Trie Diagram (By: Andrew R. Jackson)*

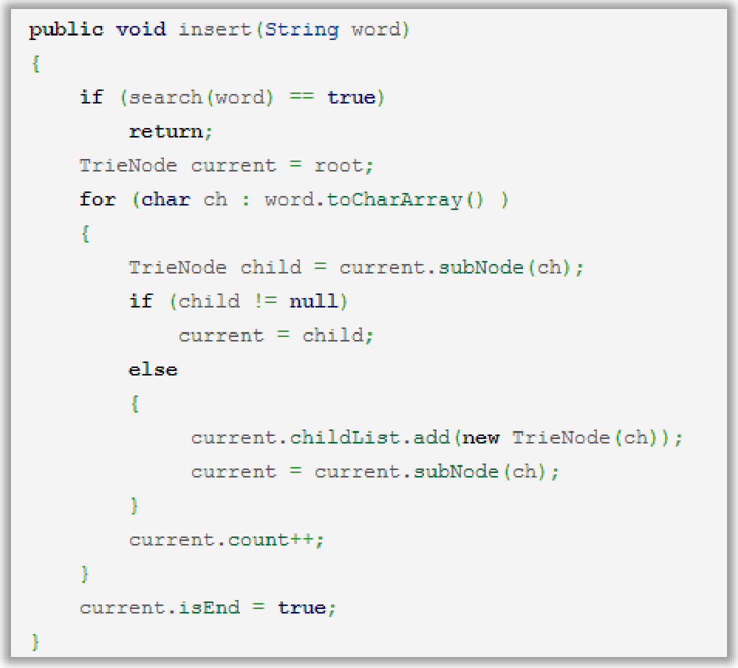
Since the directory works like an array, it provides direct access to each entry and therefore, it reaches the appropriate bucket in a single disk access within constant time. Like nodes, each bucket represent a disk page and can hold a fixed limited number of IDs. Instead of using parallel arrays, each bucket contains of a Trie tree structure that holds the address of the IDs in the leaf nodes (Healey, Extendible Hashing, 90). Every bucket has a local depth l (Zhang). This system uses the local depth to determine when the directory size needs to be double or halved.

As seen from **Figure 4**, the root node consist of one bit and therefore, the value of l is one. As the system goes down the tree, the branches divide in a binary format where the left branch would have elements that has 0 as the next bit while the right branch would have elements that have 1 as the next bit. Unlike the B-tree, this Trie is not a balanced structure because the leaf nodes are not in the same height (Jackson).

In all Extendible hashing methods/processes, the system passes the ID to the hashing function to reach the appropriate bucket where the system inserts, searches or deletes the ID. Once it reaches the bucket stored in the secondary memory, it accesses the disk and copies the bucket into RAM where it completes the task in the Trie tree structure.

3.1 Insertion

When the system passes the ID to the tree, the system moves down the tree recursively through the appropriate sub-trees until it reaches the point when the similarity between the bits of the ID and the nodes in the sub-tree ends. Then, the system creates a new leaf node and inserts the value of the ID and the address into the node. **Figure 5** is an implementation of the Insertion method in a Trie that uses Strings for IDs and the characters of the Strings instead of bits.



```java
public void insert(String word)
{
    if (search(word) == true)
        return;
    TrieNode current = root;
    for (char ch : word.toCharArray() )
    {
        TrieNode child = current.subNode(ch);
        if (child != null)
            current = child;
        else
        {
             current.childList.add(new TrieNode(ch));
             current = current.subNode(ch);
        }
        current.count++;
    }
    current.isEnd = true;
}
```

***Figure 5**: Insertion in Trie. Coded by Manish Bhojasia (www.sanfoundry.com)*

If the Trie has more values than permitted, the system splits the bucket into two. Then the system inserts the ID into the appropriate bucket. When a bucket splits, the local depth of the bucket increments. If the local depth is equal to the global depth, the directory size doubles and the global depth is incremented (Healey, Extendible Hashing, 92).

3.2 Search

Search method is simpler than the Insertion method. The system searches the Trie recursively until it finds the ID. It works like the binary tree when searching for an element.

3.3 Deletion

The first part of the deletion method is identical to the search method. The system searches for the ID recursively through the Trie. After it finds the ID, it checks if there is more than one branch in the tree. If yes, the system deletes the ID. Otherwise, it merges with the neighboring bucket and then, it deletes the ID. The local depth is then decremented (Healey, Extendible Hashing, 94).

4 Comparison: Disk Access and Efficiency

Database indexing occurs in both primary and secondary memory. The access time in primary memory is orders of magnitude shorter than the time it takes to access the secondary memory. This means that the system spends most time throughout the duration of a process to wait for the secondary memory to respond. Therefore, the factor that affects the time taken for the completion of a process is the number of time it accesses the secondary memory such as a hard disk.

A database index is good if it takes the shortest possible time to complete a task. Therefore, a database index that would make the smallest number of disk accesses in that process is the best database index for that task.

4.1 Disk Access in B-trees

When searching for a particular ID in a B-tree, in worst-case scenario the number of nodes the system reads is equal to the height of the tree because the ID searched will be in the leaf node. When the system accesses a node in the B-tree, it creates a copy in the RAM of the items in the disk page represented by the node. It then searches for the ID within the RAM. Each time it makes an access into the disk, it takes a unit time.

If for instance, the height of the B-tree is three and the value of the variable t is five, the maximum number of items that the node can store is nine and the number of nodes that each node can point is 10. Assuming that all the nodes are full, the number of nodes inside the tree are:

$$1 + 10 + 10 \times 10 = 111\,nodes$$

Since all the nodes have nine IDs, the total number of IDs in the B-tree is 999. This means that it will take three disk accesses to access an ID with 999 IDs. Here are the workings that help in determining the number of disk accesses in terms of number of keys, height of the tree and t.

$$Let\ n = number\ of\ keys, H = height\ of\ the\ tree\ and\ t = static\ variable\ described\ above$$

$$n = \sum_{x=1}^{H}((2t-1)((2t)^{x-1})$$

$$n = \frac{(2t-1)((2t)^{H}-1)}{2t-1}$$

$$n = (2t)^{H} - 1$$

$$n + 1 = (2t)^{H}$$

$$\mathrm{H} = \log_{2\mathrm{t}}(\mathrm{n}+1)$$

This calculation shows that the relation between the number of IDs and the number of disk accesses is a logarithmic relation. Therefore, the efficiency of the B-tree increases as the number of IDs increases because the efficiency of the B-tree is negatively proportional to the slope of the number of disk accesses. This shows that B-trees are database indexes designed for large databases.

4.2 Disk Access in Extendible Hash Tables

When searching for an ID in an Extendible Hash Table, in worst-case scenario the number of disk access is 1-2. If the directory is too big to be stored in the primary memory, the first disk access will be when the system accesses the directory entry stored in the secondary memory. The second disk access will be the bucket that stores the ID. Once the system reaches the bucket, it copies the contents into the RAM, where it searches for the correct ID.

This shows that the number of disk accesses performed by a system that uses Extendible Hash Tables is not affected by the size of the database because the hashing function will allow the system into directly accessing the entry in the directory and then the bucket. Since the performance is the same even when the size of the database is big, the efficiency of the bucket increases gradually. Therefore, Extendible Hash Tables are database indexes designed for databases of all sizes since it takes the same access time for all database sizes because they provide minimum time for any size.

4.3 Oxford English Corpus: B-trees or Extendible Hash Tables

As described earlier, the Oxford English Corpus stores more than 2 billion words. If the corpus uses B-trees for database indexing and the user commands the system to find a particular word, the number of disk access is:

$$\mathrm{H} = \log_{2\mathrm{t}}(2{,}000{,}000{,}000 + 1)$$

If $t = 9, H = 7.409.. \sim 8$ disk accesses.

If Extendible Hash Tables are used in this situation, the maximum number of disk access will be two. The Extendible Hash Table is four times faster than the B-tree in this scenario. This concludes that if a database created had to find a particular item, then the best option will be to use Extendible Hash Tables

Which index is a better option if the task was to find a sequence of IDs? For example, the user tasks the system to find all the words that start with the character 'C'. Are Extendible Hash Tables the better option in this situation? Apparently not.

If a B-tree is used, the system will have the IDs sorted alphabetically. The system will have to go to the sub-tree that contains all the IDs that start with a C. The number of disk accesses will depend upon the number of values that start with a C. If there are 100 million words that start with a C, the system will access 11,111,112 nodes and will perform the same number of disk accesses. Extendible Hash Tables on the other hand do not sort the IDs in such order. Therefore, they will have to check all the buckets that exist in the index. If each bucket also stores nine IDs, then the number of buckets that the system will access is 222,222,223 that is equal to the number of disk access that the system will perform. Hence, the B-trees are about 20 times faster than the Extendible Hash Tables in this task. Likewise, if the user asks the system to find all the words that start with a keyword or to provide a list of all the words in an alphabetical order, B-trees will stand better than the Extendible Hash Tables.

The two types of queries mentioned above are exact and range matches (3 Indexes and Index-Organized Tables). Exact query requires searching for an exact ID. For example, searching for the word "dog" is an exact query because the system knows exactly that the ID to be found must have three characters 'd', 'o' and 'g' and arranged in the permutation of 'dog'. In such tasks that have exact query, Extendible Hash Tables are certainly the better option. On the other hand, Range queries are tasks that require searching for a list of IDs that match a particular range. For example, searching for all words that have the prefix "pre". In this query type, B-tree is a better index than Extendible Hash Table.

5 <u>Conclusion</u>

The research question investigated: *To what extent does the structure of B-tree make it a better choice than Extendible Hash Tables for creating database indexes.*

The answer that is discovered from the investigation of the research question is that both B-trees and Extendible Hash Tables have their own merits in database indexing. Referring from the calculations above, B-trees have an upper hand in range queries compared to Extendible Hash Tables because they have lower number of disk accesses and therefore take less time to complete the query. Whereas, in exact queries, Extendible Hash Tables are the more sensible option as indexes.

However, it can be argued that B-trees are better than Extendible Hash Tables because of their versatility and the ability of having the best efficiency overall. Despite the fact that B-trees perform worse than Extendible Hash Tables in exact queries, in terms of Big-O notation, B-trees perform at $O(log\ n)$ efficiency while Extendible Hash Tables perform at $O(1)$ efficiency when dealing with exact queries (Big-O Notation). However, when performing range queries, B-trees maintain the $O(log\ n)$ efficiency whereas Extendible Hash Tables perform at $O(n)$ efficiency. This suggests that the performance of Extendible Hash Tables deteriorate if the tasks given to it switches from exact queries to range queries. Therefore, this shows that the Extendible Hash Tables are able to do only a limited set of tasks. On the other hand, B-trees maintain the high efficiency in both exact and range queries and therefore it suggests that they are more versatile and can perform a larger variety of tasks efficiently and within minimal duration.

To solve the dilemma of the programmers when designing databases, if the programmers require an index for exact queries then they must choose Extendible Hash Tables. Otherwise, if the programmers require an index for range queries or both exact and range queries, then they must choose B-trees.

What if there was a better option than B-trees. What if the successor of B-trees, B+trees, may have a higher efficiency than B-trees? B+trees store the data only in leaf nodes and not in internal and root nodes. This allows the system to store the root node and the internal nodes in the primary memory because of their small size. Technically, this would reduce the number of disk access greatly because the system would have to access the disk only to access the correct leaf node, hence performing exact, and range queries more efficiently. In that case, would the B+trees prove to be as efficient as Extendible Hash Tables, if not more, in solving exact queries? In that case, no matter what scenario a database may have and irrespective of the types of tasks to be performed, would the B+trees prove to be the optimum abstract data structure for creating database index?

Work Cited

"3 Indexes and Index-Organized Tables." *Indexes and Index-Organized Tables*. N.p., n.d. Web. 19 Nov. 2014. <https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT1170>.

Beal, Vangie. "Database." *What Is Database (DB)? Webopedia*. N.p., n.d. Web. 19 Nov. 2014. <http://www.webopedia.com/TERM/D/database.html>.

Bhojasia, Manish. "Java Program to Implement Trie." *Sanfoundry*. N.p., n.d. Web. 19 Nov. 2014. <http://www.sanfoundry.com/java-program-implement-trie/>.

"Big-O Notation." *Big-O*. N.p., n.d. Web. 19 Nov. 2014. <http://www.csee.umbc.edu/courses/undergraduate/202/spring07/Lectures/ChangSynopses/modules/m33-big-O/slides.php?print>.

"Extendible Hashing for COSC 311." *Extendible Hashing for COSC 311*. N.p., 4 Sept. 2007. Web. 19 Nov. 2014. <http://www.emunix.emich.edu/~haynes/Papers/ExtendibleHashing/extendibleHashing.html>.

Guz, Savannah S. "BEST DATABASES." *Oxford English Corpus* (2012): 6-12. *UBESCO*. Web. 16 Nov. 2014. <http://search.ebscohost.com/login.aspx?direct=true&db=ulh&AN=67670504&site=src-live>.

Healey, Christopher G. "CSC 541 Advance Data Structure." *Extendible Hashing*(2014): 89-95. North Carolina State University. Webhea. <http://www.csc.ncsu.edu/faculty/healey/csc541/notes/notes.pdf>.

Healey, Christopher G. "CSC 541 Advance Data Structure." *File Indexing* (2014): 25-31. North Carolina State University. Web. <http://www.csc.ncsu.edu/faculty/healey/csc541/notes/notes.pdf>.

Jackson, Andrew R. "DATA STRUCTURES AND ALGORITHMS Topic #16: HEAPS." McGill University, 15 Aug. 1999. Web. <http://www.cgm.cs.mcgill.ca%2F~hagha%2Ftopic16%2Ftopic16.html>.

Khan, Rizwan. "Your Dream Land." *: Awesome Apps For Students*. N.p., 12 Jan. 2014. Web. 19 Nov. 2014. <http://rizu842.blogspot.com/2014/01/awesome-apps-for-students.html>.

Neubauer, Peter. "B-Trees." *Bluer White*. N.p., n.d. Web. <http://www.bluerwhite.org%2Fbtree%2F>.

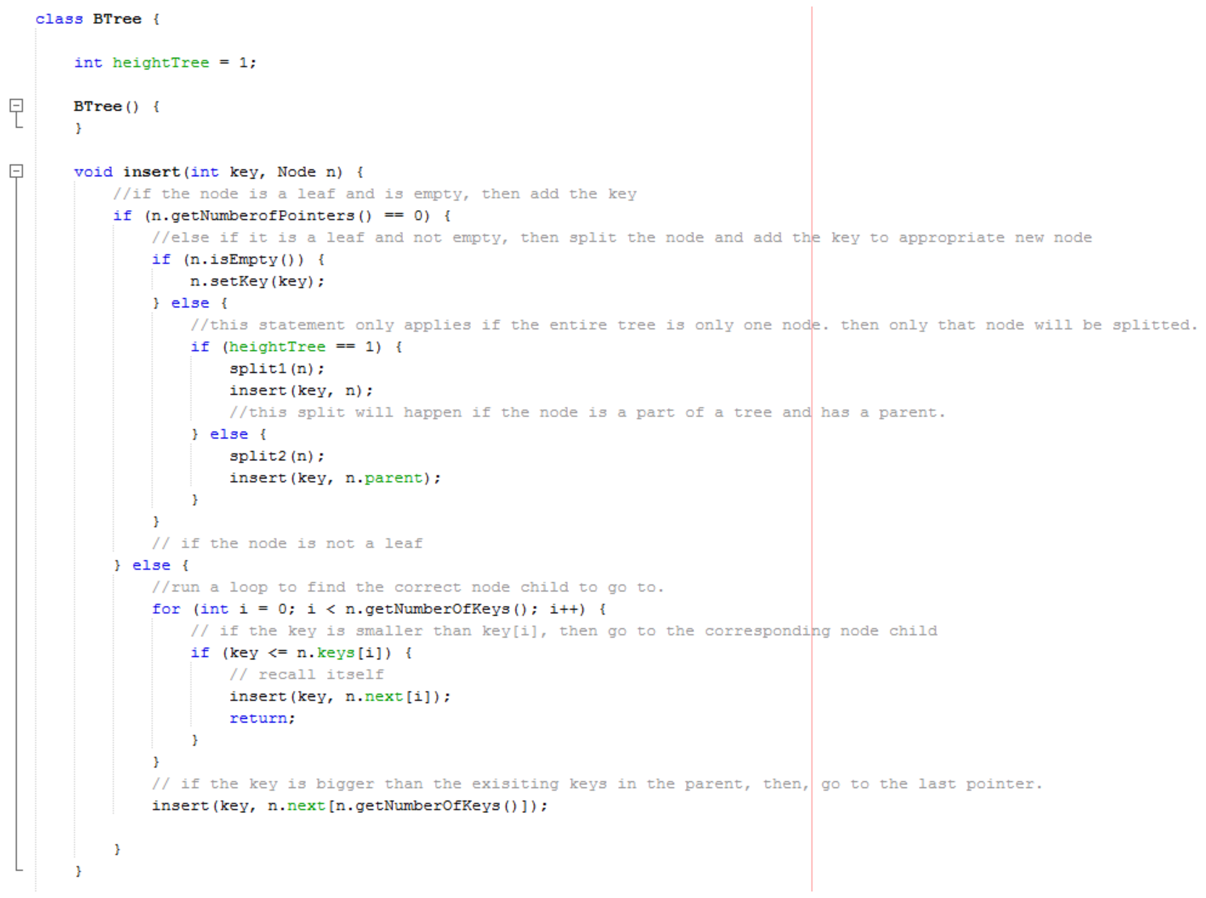
Rouse, Margaret. "Database." *What Is ?* N.p., n.d. Web. 16 Aug. 2014. <http://searchsqlserver.techtarget.com/definition/database>. “sales transactions, product catalogs and inventories”

Zang, Donghui. *B Trees*. *Handbook of Data Structures and Applications*. By Dinesh P. Mehta and Sartaj Sahni. Boca Raton, FL: Chapman & Hall/CRC, 2005. N. pag. Print. "every node corresponds to a disk page"

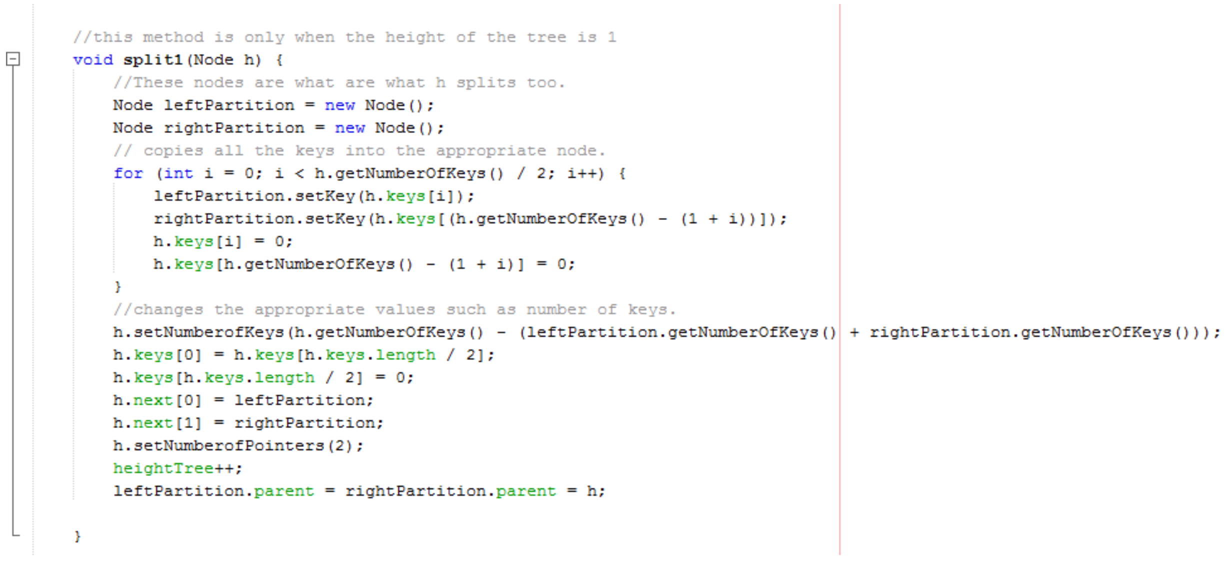
Zhang, Donghui. "Extendible Hashing." *Extendible Hashing* (n.d.): n. pag. Web. <http://zgking.com:8080/home/donghui/publications/books/e_ds_extendiblehashing.pdf>.

Appendix

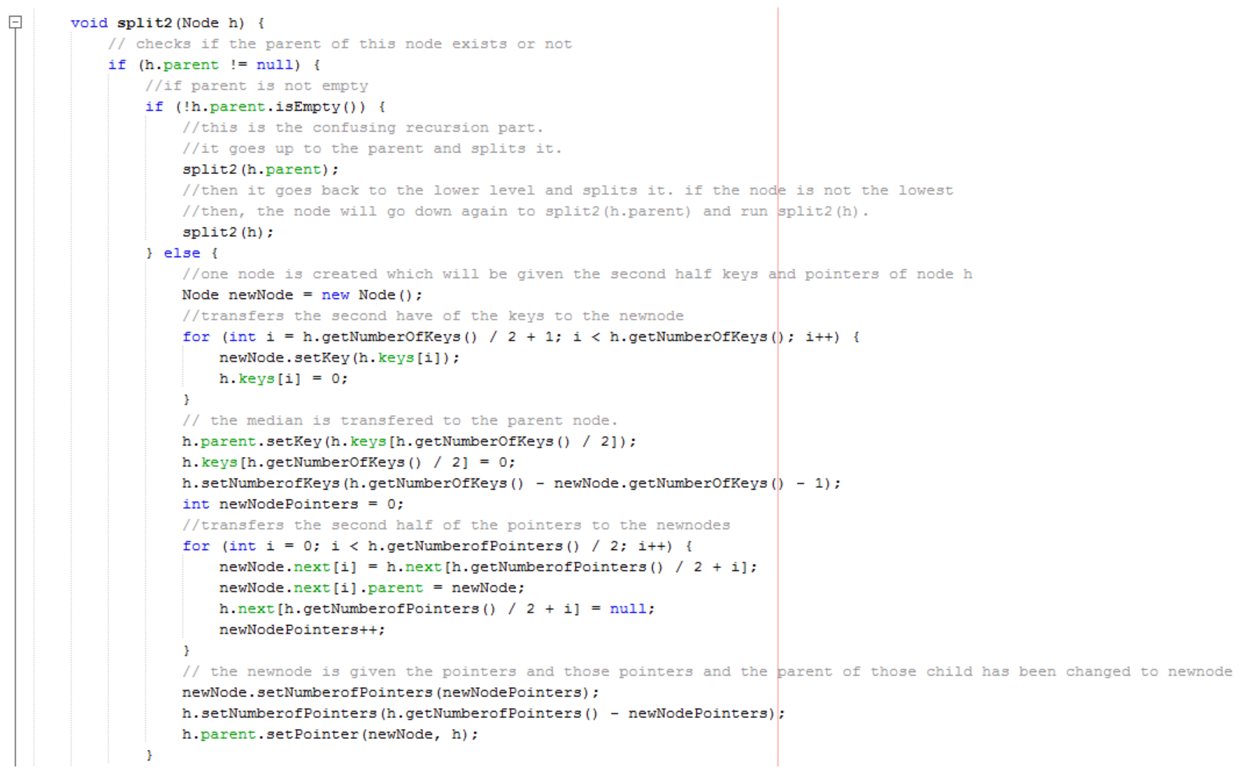
Personal Code for B-tree Data Structure



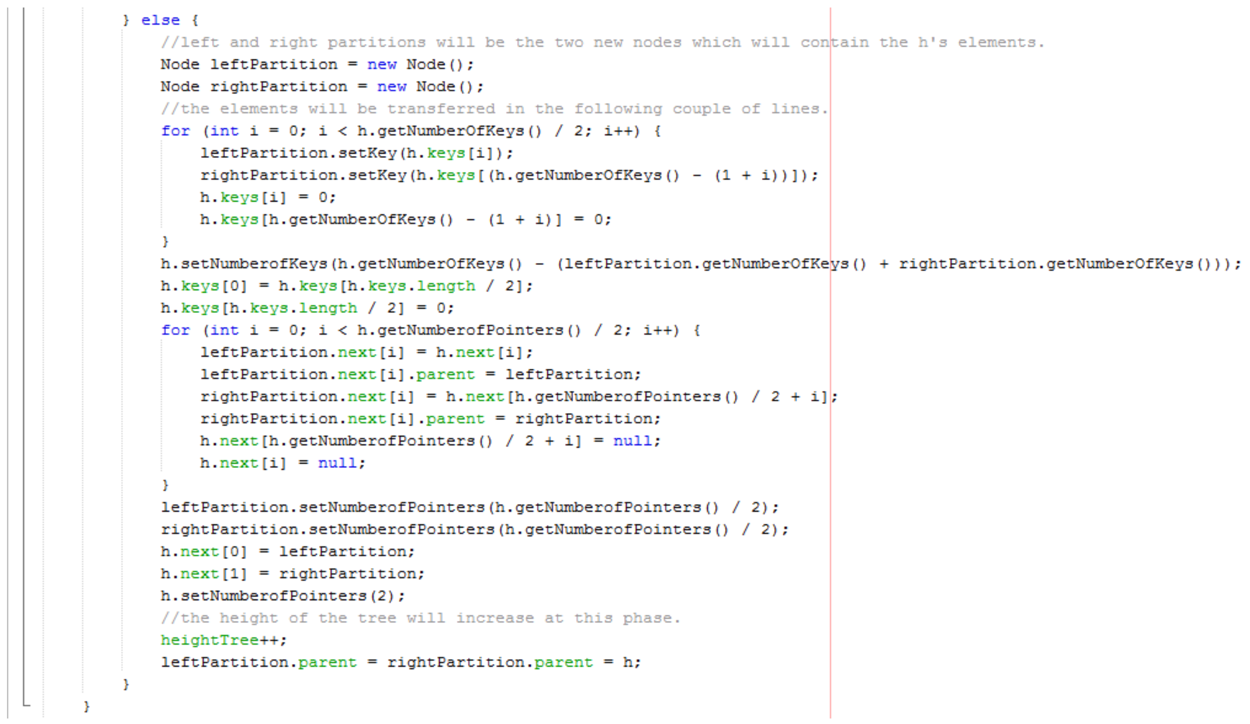
```
class BTree {

    int heightTree = 1;

    BTree() {
    }

    void insert(int key, Node n) {
        //if the node is a leaf and is empty, then add the key
        if (n.getNumberofPointers() == 0) {
            //else if it is a leaf and not empty, then split the node and add the key to appropriate new node
            if (n.isEmpty()) {
                n.setKey(key);
            } else {
                //this statement only applies if the entire tree is only one node. then only that node will be splitted.
                if (heightTree == 1) {
                    split1(n);
                    insert(key, n);
                    //this split will happen if the node is a part of a tree and has a parent.
                } else {
                    split2(n);
                    insert(key, n.parent);
                }
            }
            // if the node is not a leaf
        } else {
            //run a loop to find the correct node child to go to.
            for (int i = 0; i < n.getNumberOfKeys(); i++) {
                // if the key is smaller than key[i], then go to the corresponding node child
                if (key <= n.keys[i]) {
                    // recall itself
                    insert(key, n.next[i]);
                    return;
                }
            }
            // if the key is bigger than the exisiting keys in the parent, then, go to the last pointer.
            insert(key, n.next[n.getNumberOfKeys()]);

        }
    }
```



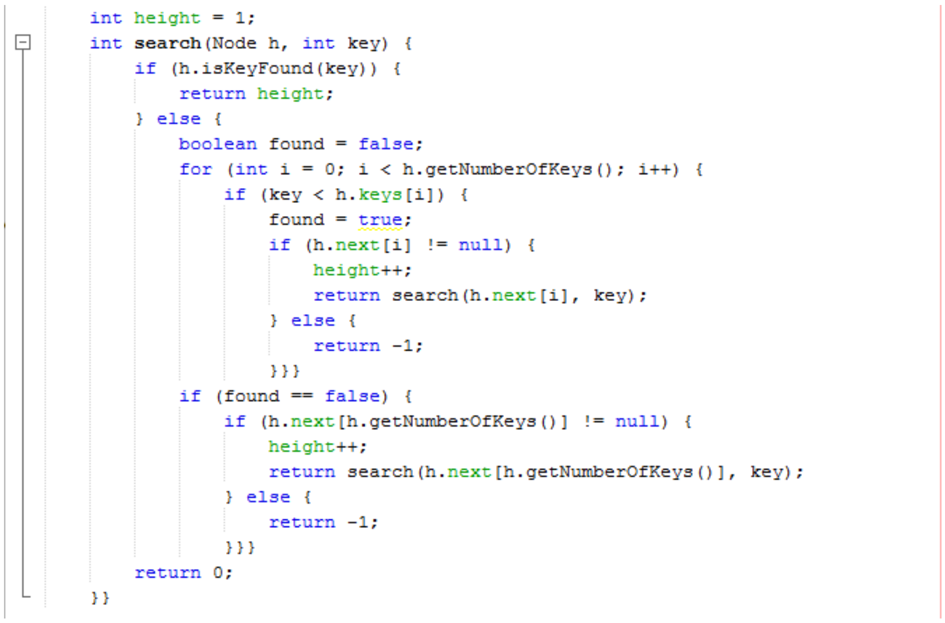
```
    //this method is only when the height of the tree is 1
    void split1(Node h) {
        //These nodes are what are what h splits too.
        Node leftPartition = new Node();
        Node rightPartition = new Node();
        // copies all the keys into the appropriate node.
        for (int i = 0; i < h.getNumberOfKeys() / 2; i++) {
            leftPartition.setKey(h.keys[i]);
            rightPartition.setKey(h.keys[(h.getNumberOfKeys() - (1 + i))]);
            h.keys[i] = 0;
            h.keys[h.getNumberOfKeys() - (1 + i)] = 0;
        }
        //changes the appropriate values such as number of keys.
        h.setNumberofKeys(h.getNumberOfKeys() - (leftPartition.getNumberOfKeys() + rightPartition.getNumberOfKeys()));
        h.keys[0] = h.keys[h.keys.length / 2];
        h.keys[h.keys.length / 2] = 0;
        h.next[0] = leftPartition;
        h.next[1] = rightPartition;
        h.setNumberofPointers(2);
        heightTree++;
        leftPartition.parent = rightPartition.parent = h;

    }
```



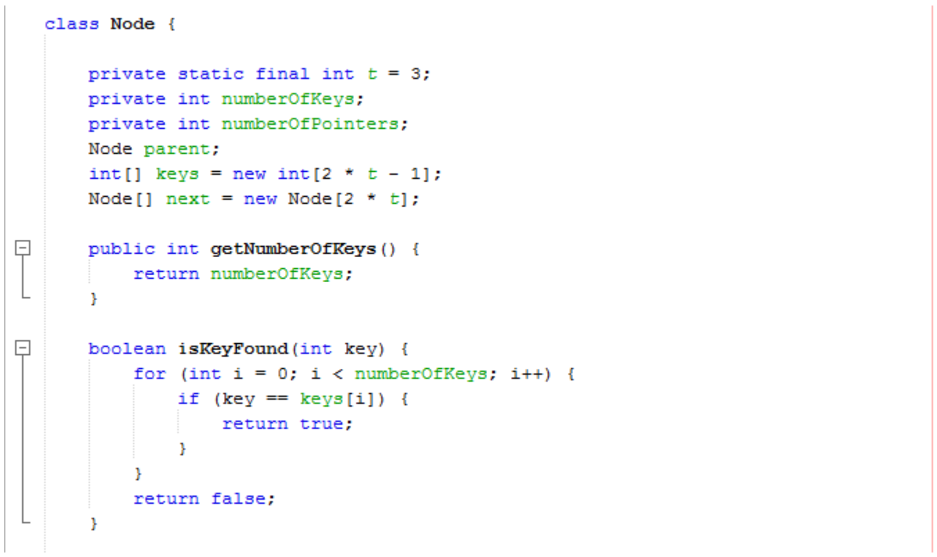
```
void split2(Node h) {
    // checks if the parent of this node exists or not
    if (h.parent != null) {
        //if parent is not empty
        if (!h.parent.isEmpty()) {
            //this is the confusing recursion part.
            //it goes up to the parent and splits it.
            split2(h.parent);
            //then it goes back to the lower level and splits it. if the node is not the lowest
            //then, the node will go down again to split2(h.parent) and run split2(h).
            split2(h);
        } else {
            //one node is created which will be given the second half keys and pointers of node h
            Node newNode = new Node();
            //transfers the second have of the keys to the newnode
            for (int i = h.getNumberOfKeys() / 2 + 1; i < h.getNumberOfKeys(); i++) {
                newNode.setKey(h.keys[i]);
                h.keys[i] = 0;
            }
            // the median is transfered to the parent node.
            h.parent.setKey(h.keys[h.getNumberOfKeys() / 2]);
            h.keys[h.getNumberOfKeys() / 2] = 0;
            h.setNumberofKeys(h.getNumberOfKeys() - newNode.getNumberOfKeys() - 1);
            int newNodePointers = 0;
            //transfers the second half of the pointers to the newnodes
            for (int i = 0; i < h.getNumberofPointers() / 2; i++) {
                newNode.next[i] = h.next[h.getNumberofPointers() / 2 + i];
                newNode.next[i].parent = newNode;
                h.next[h.getNumberofPointers() / 2 + i] = null;
                newNodePointers++;
            }
            // the newnode is given the pointers and those pointers and the parent of those child has been changed to newnode
            newNode.setNumberofPointers(newNodePointers);
            h.setNumberofPointers(h.getNumberofPointers() - newNodePointers);
            h.parent.setPointer(newNode, h);
        }
```



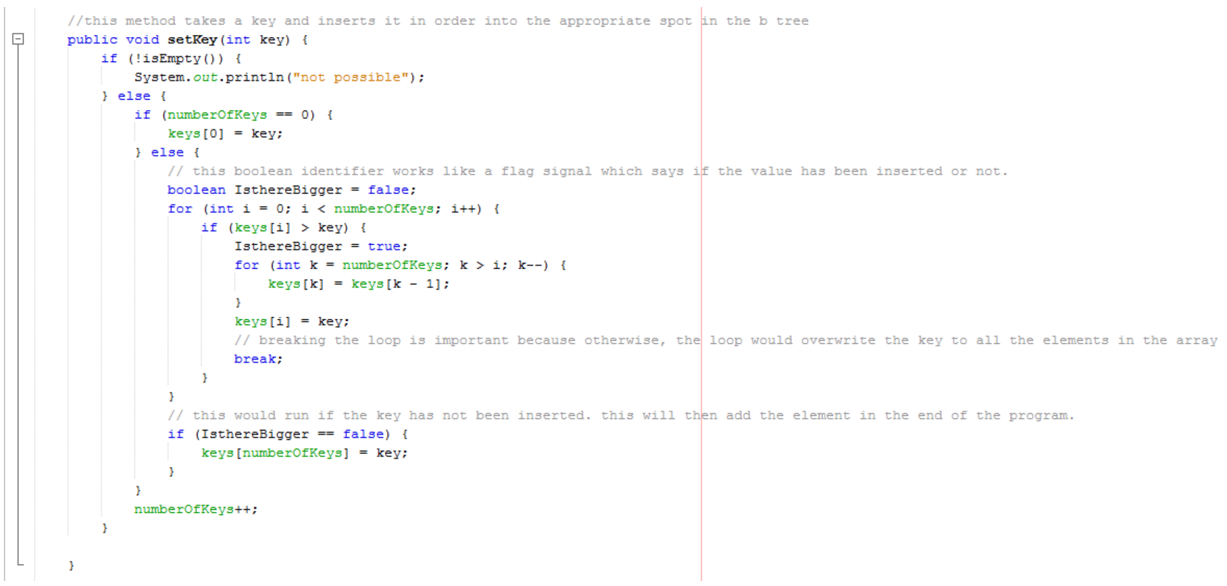
```
    } else {
        //left and right partitions will be the two new nodes which will contain the h's elements.
        Node leftPartition = new Node();
        Node rightPartition = new Node();
        //the elements will be transferred in the following couple of lines.
        for (int i = 0; i < h.getNumberOfKeys() / 2; i++) {
            leftPartition.setKey(h.keys[i]);
            rightPartition.setKey(h.keys[(h.getNumberOfKeys() - (1 + i))]);
            h.keys[i] = 0;
            h.keys[h.getNumberOfKeys() - (1 + i)] = 0;
        }
        h.setNumberofKeys(h.getNumberOfKeys() - (leftPartition.getNumberOfKeys() + rightPartition.getNumberOfKeys()));
        h.keys[0] = h.keys[h.keys.length / 2];
        h.keys[h.keys.length / 2] = 0;
        for (int i = 0; i < h.getNumberofPointers() / 2; i++) {
            leftPartition.next[i] = h.next[i];
            leftPartition.next[i].parent = leftPartition;
            rightPartition.next[i] = h.next[h.getNumberofPointers() / 2 + i];
            rightPartition.next[i].parent = rightPartition;
            h.next[h.getNumberofPointers() / 2 + i] = null;
            h.next[i] = null;
        }
        leftPartition.setNumberofPointers(h.getNumberofPointers() / 2);
        rightPartition.setNumberofPointers(h.getNumberofPointers() / 2);
        h.next[0] = leftPartition;
        h.next[1] = rightPartition;
        h.setNumberofPointers(2);
        //the height of the tree will increase at this phase.
        heightTree++;
        leftPartition.parent = rightPartition.parent = h;
    }
}
```



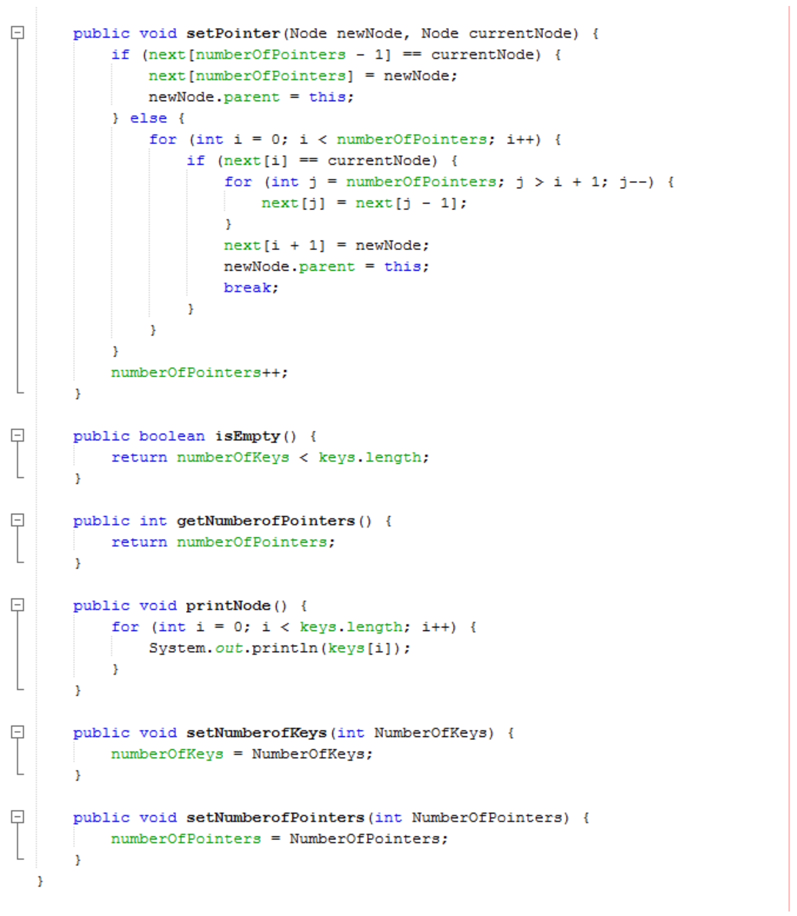
```java
    int height = 1;
    int search(Node h, int key) {
        if (h.isKeyFound(key)) {
            return height;
        } else {
            boolean found = false;
            for (int i = 0; i < h.getNumberOfKeys(); i++) {
                if (key < h.keys[i]) {
                    found = true;
                    if (h.next[i] != null) {
                        height++;
                        return search(h.next[i], key);
                    } else {
                        return -1;
                    }}}
            if (found == false) {
                if (h.next[h.getNumberOfKeys()] != null) {
                    height++;
                    return search(h.next[h.getNumberOfKeys()], key);
                } else {
                    return -1;
                }}}
        return 0;
    }}
```



```java
class Node {

    private static final int t = 3;
    private int numberOfKeys;
    private int numberOfPointers;
    Node parent;
    int[] keys = new int[2 * t - 1];
    Node[] next = new Node[2 * t];

    public int getNumberOfKeys() {
        return numberOfKeys;
    }

    boolean isKeyFound(int key) {
        for (int i = 0; i < numberOfKeys; i++) {
            if (key == keys[i]) {
                return true;
            }
        }
        return false;
    }
```

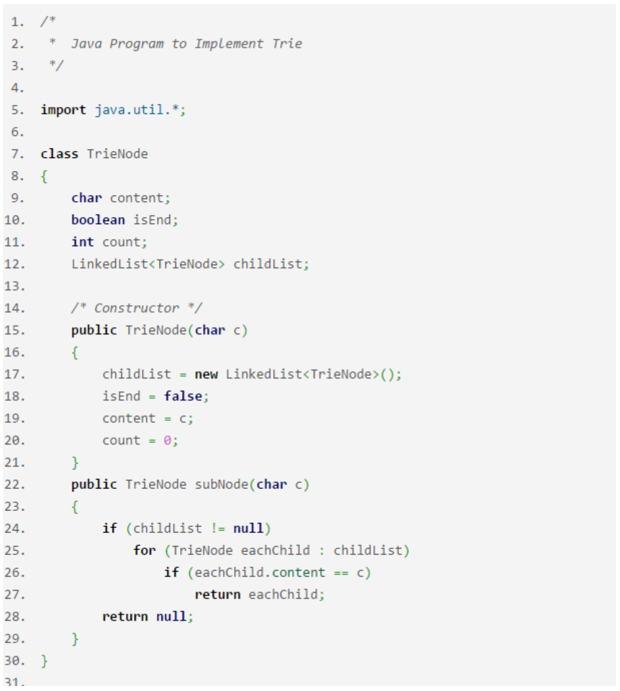


```java
//this method takes a key and inserts it in order into the appropriate spot in the b tree
public void setKey(int key) {
    if (!isEmpty()) {
        System.out.println("not possible");
    } else {
        if (numberOfKeys == 0) {
            keys[0] = key;
        } else {
            // this boolean identifier works like a flag signal which says if the value has been inserted or not.
            boolean IsthereBigger = false;
            for (int i = 0; i < numberOfKeys; i++) {
                if (keys[i] > key) {
                    IsthereBigger = true;
                    for (int k = numberOfKeys; k > i; k--) {
                        keys[k] = keys[k - 1];
                    }
                    keys[i] = key;
                    // breaking the loop is important because otherwise, the loop would overwrite the key to all the elements in the array
                    break;
                }
            }
            // this would run if the key has not been inserted. this will then add the element in the end of the program.
            if (IsthereBigger == false) {
                keys[numberOfKeys] = key;
            }
        }
        numberOfKeys++;
    }

}
```

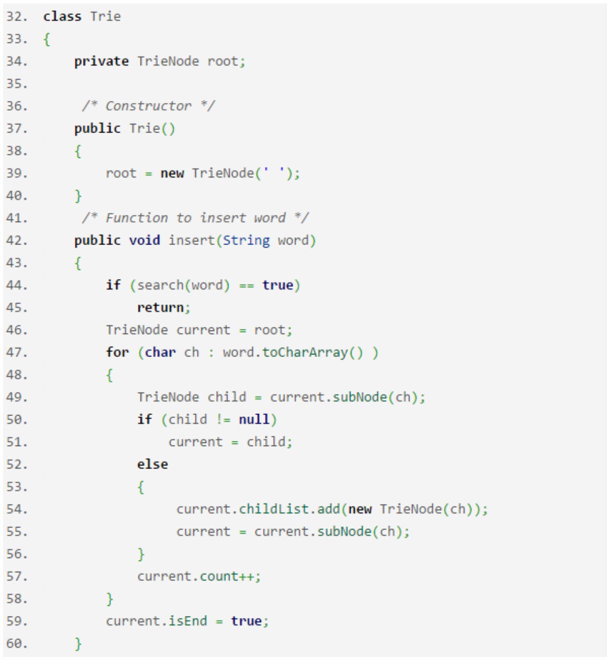


```java
    public void setPointer(Node newNode, Node currentNode) {
        if (next[numberOfPointers - 1] == currentNode) {
            next[numberOfPointers] = newNode;
            newNode.parent = this;
        } else {
            for (int i = 0; i < numberOfPointers; i++) {
                if (next[i] == currentNode) {
                    for (int j = numberOfPointers; j > i + 1; j--) {
                        next[j] = next[j - 1];
                    }
                    next[i + 1] = newNode;
                    newNode.parent = this;
                    break;
                }
            }
        }
        numberOfPointers++;
    }

    public boolean isEmpty() {
        return numberOfKeys < keys.length;
    }

    public int getNumberofPointers() {
        return numberOfPointers;
    }

    public void printNode() {
        for (int i = 0; i < keys.length; i++) {
            System.out.println(keys[i]);
        }
    }

    public void setNumberofKeys(int NumberOfKeys) {
        numberOfKeys = NumberOfKeys;
    }

    public void setNumberofPointers(int NumberOfPointers) {
        numberOfPointers = NumberOfPointers;
    }
}
```

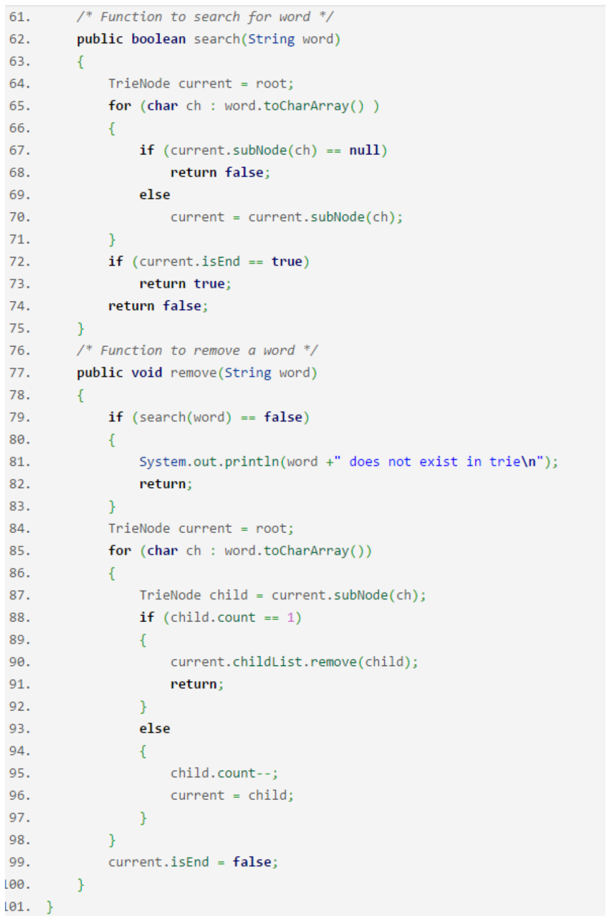
Outsourced Code for Trie Data Structure:



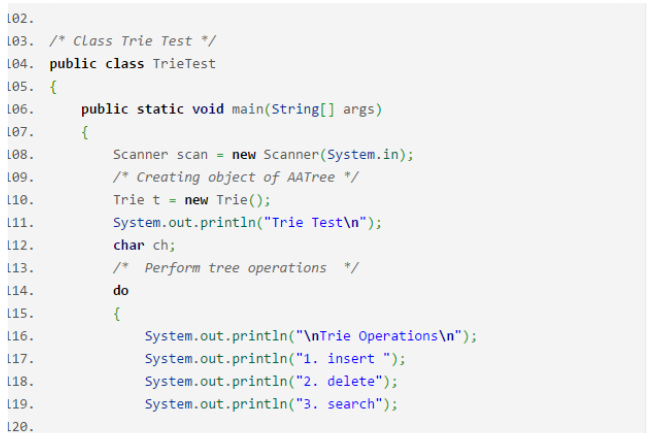
```java
/*
 *  Java Program to Implement Trie
 */

import java.util.*;

class TrieNode
{
    char content;
    boolean isEnd;
    int count;
    LinkedList<TrieNode> childList;

    /* Constructor */
    public TrieNode(char c)
    {
        childList = new LinkedList<TrieNode>();
        isEnd = false;
        content = c;
        count = 0;
    }
    public TrieNode subNode(char c)
    {
        if (childList != null)
            for (TrieNode eachChild : childList)
                if (eachChild.content == c)
                    return eachChild;
        return null;
    }
}

```



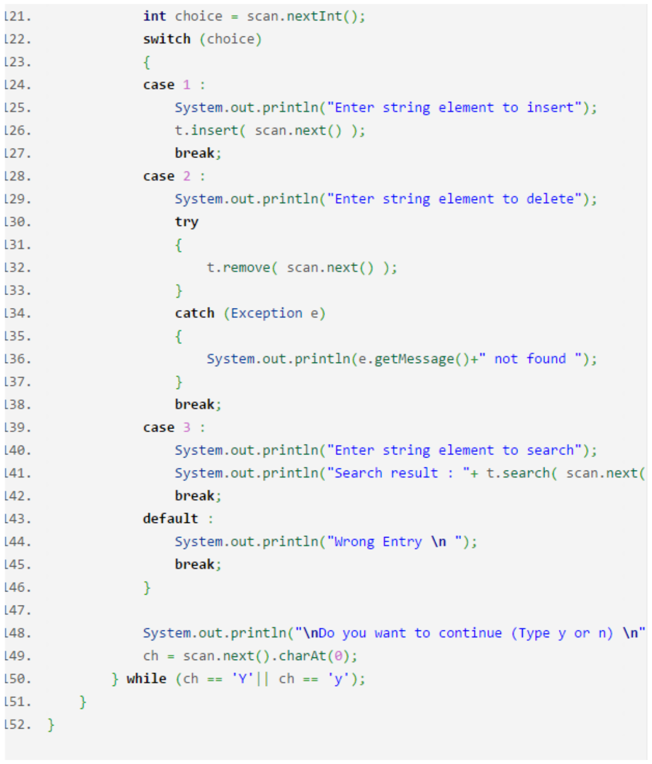
```java
class Trie
{
    private TrieNode root;

     /* Constructor */
    public Trie()
    {
        root = new TrieNode(' ');
    }
     /* Function to insert word */
    public void insert(String word)
    {
        if (search(word) == true)
            return;
        TrieNode current = root;
        for (char ch : word.toCharArray() )
        {
            TrieNode child = current.subNode(ch);
            if (child != null)
                current = child;
            else
            {
                 current.childList.add(new TrieNode(ch));
                 current = current.subNode(ch);
            }
            current.count++;
        }
        current.isEnd = true;
    }
```



```java
    /* Function to search for word */
    public boolean search(String word)
    {
        TrieNode current = root;
        for (char ch : word.toCharArray() )
        {
            if (current.subNode(ch) == null)
                return false;
            else
                current = current.subNode(ch);
        }
        if (current.isEnd == true)
            return true;
        return false;
    }
    /* Function to remove a word */
    public void remove(String word)
    {
        if (search(word) == false)
        {
            System.out.println(word +" does not exist in trie\n");
            return;
        }
        TrieNode current = root;
        for (char ch : word.toCharArray())
        {
            TrieNode child = current.subNode(ch);
            if (child.count == 1)
            {
                current.childList.remove(child);
                return;
            }
            else
            {
                child.count--;
                current = child;
            }
        }
        current.isEnd = false;
    }
}
```



```java

/* Class Trie Test */
public class TrieTest
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        /* Creating object of AATree */
        Trie t = new Trie();
        System.out.println("Trie Test\n");
        char ch;
        /*  Perform tree operations  */
        do
        {
            System.out.println("\nTrie Operations\n");
            System.out.println("1. insert ");
            System.out.println("2. delete");
            System.out.println("3. search");

```



```
            int choice = scan.nextInt();
            switch (choice)
            {
            case 1 :
                System.out.println("Enter string element to insert");
                t.insert( scan.next() );
                break;
            case 2 :
                System.out.println("Enter string element to delete");
                try
                {
                    t.remove( scan.next() );
                }
                catch (Exception e)
                {
                    System.out.println(e.getMessage()+" not found ");
                }
                break;
            case 3 :
                System.out.println("Enter string element to search");
                System.out.println("Search result : "+ t.search( scan.next(
                break;
            default :
                System.out.println("Wrong Entry \n ");
                break;
            }

            System.out.println("\nDo you want to continue (Type y or n) \n"
            ch = scan.next().charAt(0);
        } while (ch == 'Y'|| ch == 'y');
    }
}
```