Visvesvaraya Technological University

Belagavi-590 018, Karnataka

A Mini Project Report on

"Implementation of B-Tree on Hospital Management"

Mini Project Report submitted in partial fulfilment of the requirement for

the File Structures Lab [17ISL68]

Bachelor of Engineering

In

Information Science and Engineering

Submitted by

Divya.P [1JT17IS008]

Under the Guidence of

Mr.Vadiraja A

Asst. Professor Dept. Of ISE

Department of Information Science and Engineering

Jyothy Institute of Technology

Tataguni, Bengaluru-560082

2019-2020

Jyothy Institute of Technology

Tataguni, Bengaluru-560082

Department of Information Science and Engineering

CERTIFICATE

Certified that the mini project work entitled **"Implementation of B-tree"** carried out **by Divya.P [1JT17IS008]** bonfire student of Jyothy Institute of Technology, in partial fulfilment for the award of **Bachelor of Engineering** in **Information Science and Engineering** department of the **Visvesvaraya Technological University, Belagavi** during the year **2019-2020**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Degree.

Mr . Vadiraja A	Dr. Harshvardhan Tiwari
Guide, Asst. Professor	Associate. Professor and HOD
Dept. Of ISE	Dept. Of ISE

External Viva Examiner　　　　Signature with Date:

1.
2.

ACKNOWLEDGEMENT

Firstly, we are very grateful to this esteemed institution **“Jyothy Institute of Technology**” for providing us an opportunity to complete our project.

We express our sincere thanks to our Principal **Dr. Gopalakrishna K for** providing us with adequate facilities to undertake this project.

We would like to thank **Dr. Harshvardhan Tiwari , Associate Prof. and Head** of Information Science and Engineering Department for providing for his valuable support.

We would like to thank our guides **Mr.Vadiraja A** , **Asst. Prof.** for their keen interest and guidance in preparing this work.

Finally, we would thank all our friends who have helped us directly or indirectly in this project.

Divya.P[1JT17IS008]

ABSTRACT

A **B-tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. Unlike self –balancing binary search tree the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in databases and file systems. Lookup, insertion, and deletion all take O (log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation.

The term **B-tree** may refer to a specific design or it may refer to a general class of designs. In the narrow sense, a B-tree stores keys in its internal nodes but need not store those keys in the records at the leaves. The general class includes variations such as the B+ tree and the B^* tree.

Implementation of B-tree for Photography Record, id is considered as the primary key. The B-tree has been constructed for one lakh records by considering the primary key. Basically we are implementing the operations of B-tree such as insertion and search. Generation of B-tree and display of B-tree is also been implemented as part of the mini project. Time analysis is also computed for insertion, search and generation of B-tree. A time analysis graph has been plot for generation of a B-tree by considering time taken for generation of B-tree for every ten thousands of records.

Table of Contents

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

1.1 Introduction to File Structure

- A disk's relatively slow access time and the enormous, nonvolatile capacity is the driving force behind FILE STRUCTURE design!!
- FS should give access to all the capacity without making the application spend a lot of time waiting for the disk.
- FS is a combination of representation for data in files and of operations for accessing the data.
  - It allows applications to read, write and modify data
  - Also finding the data
  - Or reading the data in a particular order
- Efficiency of FS design for a particular application is decided on,
  - Details of the representation of the data
  - Implementation of the operations
- A large variety in the types of data and in the needs of application makes FS design important.
- What is best for one situation may be terrible for other.

1.2 Introduction to JAVA

- **Java** is a general-purpose computer-programming language that is concurrent, class-based, object oriented and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

  .
- Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of computer architecture.

- James Gosling, Mike Sheridan, and Patrick Noughton initiated the Java language project in June 1991. Java was originally designed for interactive television, but it was too advanced for the digital cable television industry at the time.The language was initially called *Oak* after an oak tree that stood outside Gosling's office. Later the project went by the name *Green* and was finally renamed *Java*, from Java Coffee. Gosling designed Java with a C/C++-style syntax that system and application programmers would find familiar. Sun Microsystems released the first public implementation as Java 1.0 in 1996. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms.

- There were five primary goals in the creation of the Java language:

    - It must be "simple, object-oriented, and familiar".
    - It must be "robust and secure".
    - It must be "architecture-neutral and portable".
    - It must execute with "high performance".
    - It must be "interpreted, threaded, and dynamic"

1.3 Introduction to B -tree

- A **B-tree** is a self-balancing tree data structure.

- In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation.

- Each internal node of a B-tree contains a number of keys. The keys act as separation values which divide its subtrees.

- Usually, the number of keys is chosen to vary between and, where the minimum is number the of keys, and is the minimum degree or branching factor of the tree. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has keys, then adding a key to that node can be accomplished by splitting the hypothetical key node into two key nodes and moving the key that would have been in the middle to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbour each have keys, then a key may be deleted from the internal node by combining it with its neighbour. Deleting the key would make the internal node have keys; joining the neighbour would add keys plus one more key brought down from the neighbour's parent. The result is an entirely full node of keys.

- The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters — or simply with the highest branching order,

- A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

- B-trees have substantial advantages over alternative implementations when the time to access the data of a node greatly exceeds the time spent processing that data, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the node data are in secondary storage such as disk drives. By maximizing the number of keys within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing of the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk block or an analogous size in secondary storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage need a large number of child nodes to improve performance.

CHAPTER 2

ALGORITHM

2.BASIC OPERATIONS ON B-TREES

In this section, we present the details of the operations B-TREE-SEARCH, B-TREE-CREATE, and B-TREE-INSERT. In these procedures, we adopt two conventions:

- The root of the B-tree is always in main memory, so that a DISK-READ on the root is never required; a DISK-WRITE of the root is required, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a DISK-READ operation performed on them.
- The procedures we present are all "one-pass" algorithms that proceed downward from the root of the tree, without having to back up.

2.1 Searching

Searching a B-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multiway branching decision according to the number of the node's children. More precisely, at each internal node x, we make an $(n[x] + 1)$-way branching decision.

B-TREE-SEARCH is a straightforward generalization of the TREE-SEARCH procedure defined for binary search trees. B-TREE-SEARCH takes as input a pointer to the root node x of a subtree and a key k to be searched for in that subtree. The top-level call is thus of the form B-TREE-SEARCH($root[T]$, k). If k is in the B-tree, B-TREE-SEARCH returns the ordered pair (y,i) consisting of a node y and an index i such that $key_i[y] = k$. Otherwise, the value NIL is returned.B-TREE-SEARCH(x, k)

```
1  $i \leftarrow 1$
2  while $i \leq n[x]$ and $k \geq key_i[x]$
3       do $i \leftarrow i + 1$
4  if $i \leq n[x]$ and $k = key_i[x]$
5      then return $(x, i)$
6  if $leaf[x]$
7      then return NIL
8      else DISK-READ ($c_i[x]$)
```

```
9       return B-TREE-SEARCH($c_i[x]$, $k$)
```

Using a linear-search procedure, lines 1-3 find the smallest *i* such that $k \le key_i[x]$, or else they set *i* to $n[x] + 1$. Lines 4-5 check to see if we have now discovered the key, returning if we have. Lines 6-9 either terminate the search unsuccessfully (if *x* is a leaf) or recurse to search the appropriate subtree of *x*, after performing the necessary DISK-READ on that child.

Figure shown below illustrates the operation of B-TREE-SEARCH; the lightly shaded nodes are examined during a search for the key *R*.

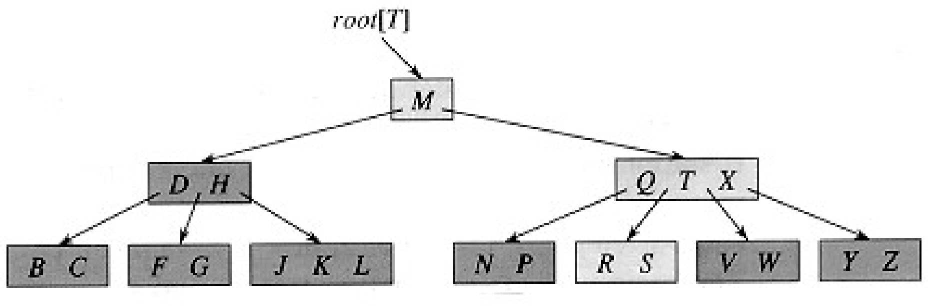


Fig 2.1 The operation of B-Tree Search

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a path downward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $\Theta(h) = \Theta(\log n)$, where *h* is the height of the B-tree and *n* is the number of keys in the B-tree. Since $n[x] < 2t$, the time taken by the **while** loop of lines 2-3 within each node is $O(t)$, and the total CPU time is $O(th) = O(t \log n)$.

2.2 Creating an empty B-tree

To build a B-tree *T*, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in $O(1)$ time. We can assume that a node created by ALLOCATE-NODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE (T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4 DISK-WRITE(x)
5  root [T] ← x
```

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

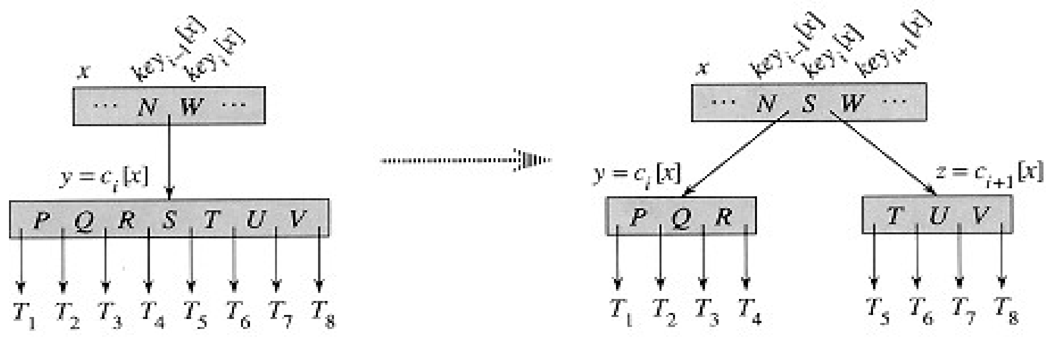


Fig2.2 Splitting a node with t = 4. Node y is split into two nodes, y and z, and the median key S of y is moved up into y's parent.

2.3 Splitting a node in a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. A fundamental operation used during insertion is the ***splitting*** of a full node y (having $2t - 1$ keys) around its ***median key*** $key^t[y]$ into two nodes having $t - 1$ keys each. The median key moves up into y's parent--which must be nonfull prior to the splitting of y--to identify the dividing point between the two new trees; if y has no parent, then the tree grows in height by one. Splitting, then, is the means by which the tree grows.

The procedure B-TREE-SPLIT-CHILD takes as input a *nonfull* internal node x (assumed to be in main memory), an index i, and a node y such that $y = c_i[x]$ is a *full* child of x. The procedure then splits this child in two and adjusts x so that it now has an additional child.

Figure 2.1 illustrates this process. The full node y is split about its median key S, which is moved up into y's parent node x. Those keys in y that are greater than the median key are placed in a new node z, which is made a new child of x.

```
B-TREE-SPLIT-CHILD(x,i,y)
1  z ← ALLOCATE-NODE ()
2  leaf[z] ← leaf[y]
3  n[z] ← t - 1
4  for j ← 1 to t - 1
5       do key^j[z] ← key^j_+^t[y]
6  if not leaf [y]
7     then for j ← 1 to t
8              do c_j[z] ← c_{j+t}[y]
9  n[y] ← t - 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_i+1[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE (z)
19  DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD works by straightforward "cutting and pasting". Here, *y* is the *i*th child of *x* and is the node being split. Node *y* originally has 2*t* - 1 children but is reduced to *t* - 1 children by this operation. Node *z* "adopts" the *t* - 1 largest children of *y*, and *z* becomes a new child of *x*, positioned just after y in *x*'s table of children. The median key of *y* moves up to become the key in *x* that separates *y* and *z*.

Lines 1-8 create node *z* and give it the larger *t* - 1 keys and corresponding *t* children of *y*. Line 9 adjusts the key count for *y*. Finally, lines 10-16 insert *z* as a child of *x*, move the median key from *y* up to *x* in order to separate *y* from *z*, and adjust *x*'s key count. Lines 17-19 write out all modified disk pages. The CPU time used by B-TREE-SPLIT-CHILD is Θ (*t*), due to the loops on lines 4-5 and 7-8. (The other loops run for at most *t* iterations.)

2.4 Inserting a key into a B-tree

Inserting a key *k* into a B-tree *T* of height *h* is done in a single pass down the tree, requiring *O(h)* disk accesses. The CPU time required is $O(th) = O(t \log n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

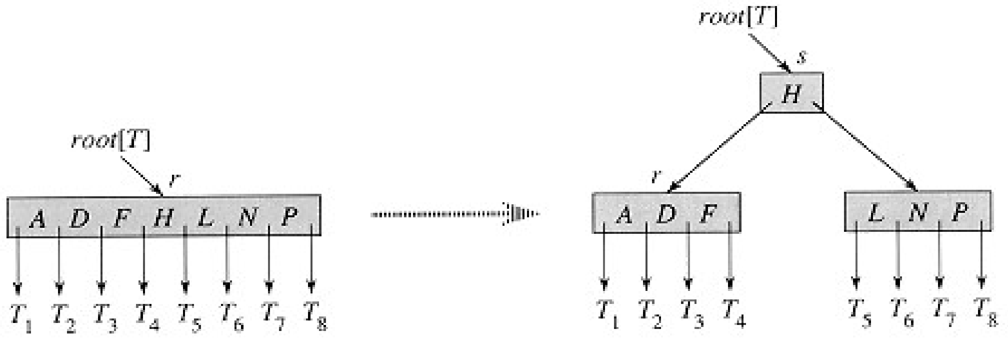


Figure 2.3 Splitting the root with t = 4. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

```
B-TREE-INSERT(T,k)
1  r ← root [T]
2  if n[r] = 2t - 1
3      then s ← ALLOCATE-NODE()
4           root[T] ← s
5           leaf[s] ← FALSE
6           n[s] ← 0
7           c^1[s] ← r
8           B-TREE-SPLIT-CHILD(s,1,r)
9           B-TREE-INSERT-NONFULL(s,k)
10 else B-TREE-INSERT-NONFULL(r,k)
```

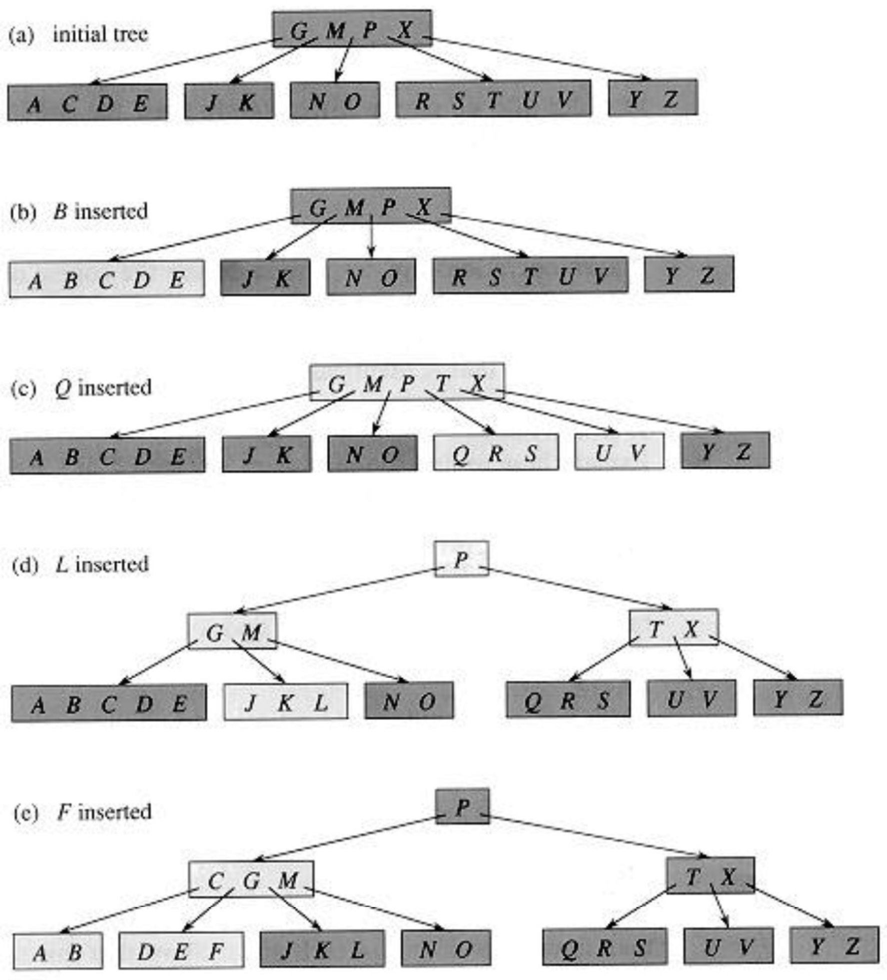
Lines 3-9 handle the case in which the root node *r* is full: the root is split and a new node *s* (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. Figure 2.2 illustrates this case. Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom. The procedure finishes by calling B-TREE-INSERT-NONFULL to perform the insertion of key *k*in the tree rooted at the nonfull root node. B-TREE-INSERT-NONFULL recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling B-TREE-SPLIT-CHILD as necessary.

The auxiliary recursive procedure B-TREE-INSERT-NONFULL inserts key *k* into node x, which is assumed to be nonfull when the procedure is called. The operation of B-TREE-INSERT and the recursive operation of B-TREE-INSERT-NONFULL guarantee that this assumption is true.

```
B-TREE-INSERT-NONFULL($x,k$)
1  $i \leftarrow n[x]$
2  if $leaf[x]$
3     then while $i \geq 1$ and $k < key^i[x]$
4              do $key_i$+1 $[x] \leftarrow key^i[x]$
5                 $i \leftarrow i - 1$
6          $key_i$+1$[x] \leftarrow k$
7          $n[x] \leftarrow$ n[x] + 1
8          DISK-WRITE($x$)
9  else while $i \geq 1$ and $k < key^i[x]$
10             do $i \leftarrow$ i - 1
11      $i \leftarrow i + 1$
12      DISK-READ ($c^i[x]$)
13      if $n[c^i[x]] = 2t - 1$
14         then B-TREE-SPLIT-CHILD($x,i,c^i[x]$)
15              if $k > key^i[x]$
16                 then $i \leftarrow i + 1$
17      B-TREE-INSERT-NONFULL($c^i[x],k$)
```

The B-TREE-INSERT-NONFULL procedure works as follows. Lines 3-8 handle the case in which x is a leaf node by inserting key k into x. If x is not a leaf node, then we must insert k into the appropriate leaf node in the subtree rooted at internal node x. In this case, lines 9-11 determine the child of x to which the recursion descends. Line 13 detects whether the recursion would descend to a full child, in which case line 14 uses B-TREE-SPLIT-CHILD to split that child into two nonfull children, and lines 15-16 determine which of the two children is now the correct one to descend to. (Note that there is no need for a DISK-READ ($c_i[x]$) after line 16 increments i, since the recursion will descend in this case to a child that was just created by B-TREE-SPLIT-CHILD.) The net effect of lines 13-16 is thus to guarantee that the procedure never recurses to a full node. Line 17 then recurses to insert k into the appropriate subtree. Figure 2.3 illustrates the various cases of inserting into a B-tree.

The number of disk accesses performed by B-TREE-INSERT is $O(h)$ for a B-tree of height h, since only $O(1)$ DISK-READ and DISK-WRITE operations are performed between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t1\text{og}_t n)$. Since B-TREE-INSERT-NONFULL is tail-recursive, it can be alternatively implemented as a **while** loop, demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.



e). Figure 2.4 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. (a) The initial tree for this example. (b) The result of inserting B into the initial tree; this is a simple insertion into a leaf node. (c) The result of inserting Q into the previous tree. The node RSTUV is split into two nodes containing RS and UV, the key T is moved up to the root, and Q is inserted in the leftmost of the two halves (the RS node). (d) The result of inserting L into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. (e) The result of inserting F into the previous tree. The node ABCDE is split before F is inserted into the rightmost of the two halves (the DE nod

2.5 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated. We sketch how it works instead of presenting the complete pseudocode.

Assume that procedure B-TREE-DELETE is asked to delete the key *k* from the subtree rooted at *x*. This procedure is structured to guarantee that whenever B-TREE-DELETE is called recursively on a node *x*, the number of keys in *x* is at least the minimum degree *t*.

Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). The following specification for deletion from a B-tree should be interpreted with the understanding that if it ever happens that the root node *x* becomes an internal node having no keys, then *x* is deleted and *x*'s only child $c_1[x]$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

Figure 2.4 illustrates the various cases of deleting keys from a B-tree.

1. If the key *k* is in node *x* and *x* is a leaf, delete the key *k* from *x*.

2. If the key *k* is in node *x* and *x* is an internal node, do the following.

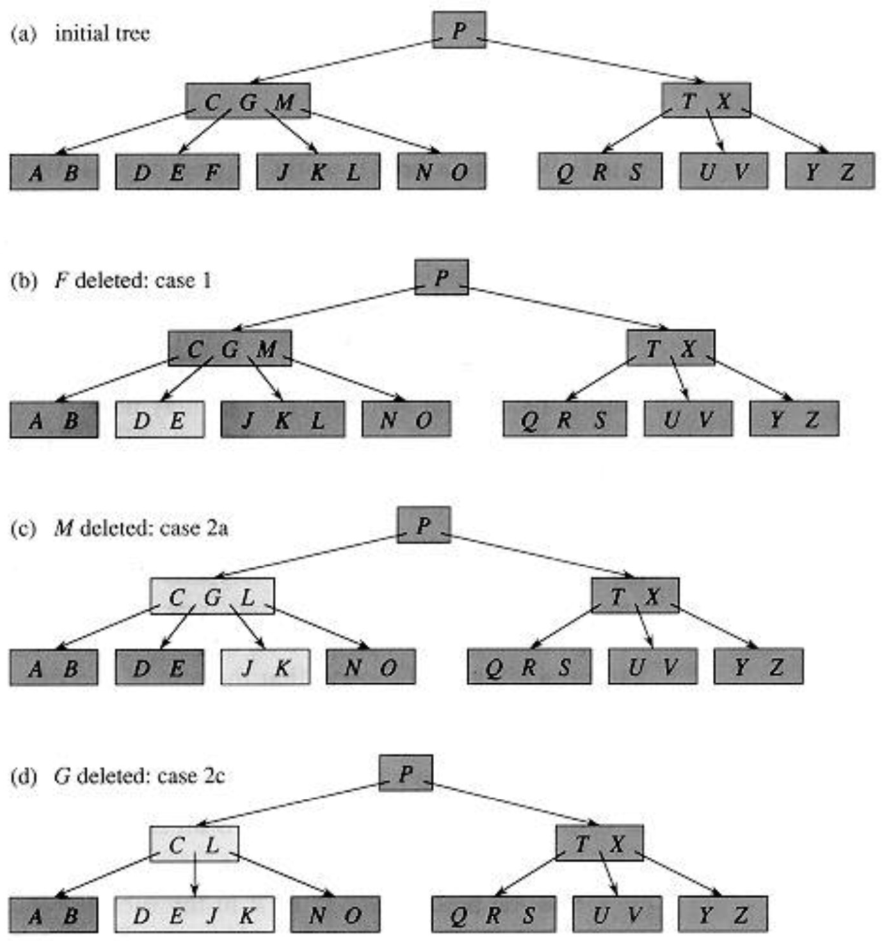
a. If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k*' of *k* in the subtree rooted at *y*. Recursively delete *k*', and replace *k* by *k*' in *x*. (Finding *k*' and deleting it can be performed in a single downward pass.)

b. Symmetrically, if the child *z* that follows *k* in node *x* has at least *t* keys, then find the successor *k*' of *k* in the subtree rooted at *z*. Recursively delete *k*', and replace *k* by *k*' in *x*. (Finding *k*' and deleting it can be performed in a single downward pass.)

c. Otherwise, if both *y* and *z* have only *t*- 1 keys, merge *k* and all of *z* into *y*, so that *x* loses both *k* and the pointer to *z*, and *y* now contains 2*t* - 1 keys. Then, free *z* and recursively delete *k* from *y*.

3. If the key *k* is not present in internal node *x*, determine the root $c_i[x]$ of the appropriate subtree that must contain *k*, if *k* is in the tree at all. If $c_i[x]$ has only *t* - 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least *t* keys. Then, finish by recursing on the appropriate child of *x*.

a. If $c_i[x]$ has only *t* - 1 keys but has a sibling with *t* keys, give $c_i[x]$ an extra key by moving a key from *x* down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into *x*, and moving the appropriate child from the sibling into $c_i[x]$.



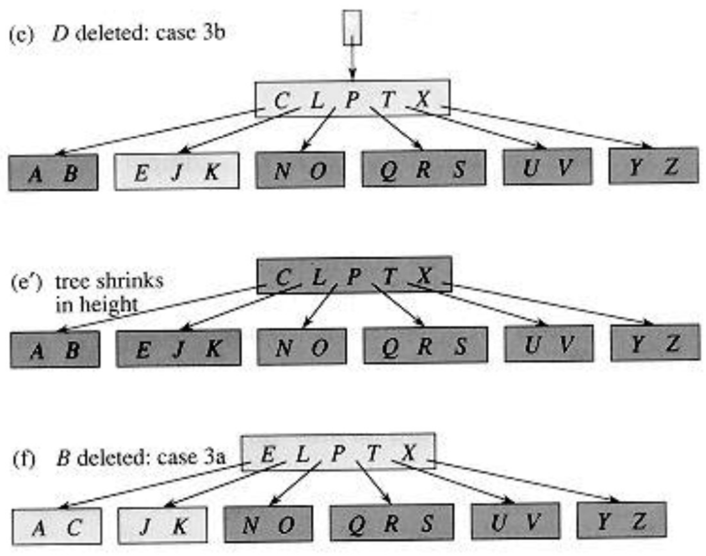


Figure 2.5 Deleting keys from a B-tree. The minimum degree for this B-tree is t = 3, so a node (other than the root) cannot have less than 2 keys. Nodes that are modified are lightly shaded. (a) The B-tree of Figure 2.4(e). (b) Deletion of F. This is case 1: simple deletion from a leaf. (c) Deletion of M. This is case 2a: the predecessor L of M is moved up to take M's position. (d) Deletion of G. This is case 2c: G is pushed down to make node DEGJK, and then G is deleted from this leaf (case 1). (e) Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form CLPTX; then, D is deleted from a leaf (case 1). (e') After (d), the root is deleted and the tree shrinks in height by one. (f) Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.

b. If $c_i[x]$ and all of $c_i[x]$'s siblings have t - 1 keys, merge c_i with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in practice, deletion operations are most often used to delete keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log n)$.

CHAPTER 3
B-TREE ANALYSIS

3. B-TREE ANALYSIS

The asymptotic cost of search, insertion, and deletion of records from B-trees, B+ trees, and B∗ trees is Θ(logn) where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B+ tree of order 100 and leaf nodes that contain up to 100 records. A B+ tree with height one (that is, just a single leaf node) can have at most 100 records. A B- B+ tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B+ tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B+ tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B+ tree of more than height four.

The B+ tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about 3/4 full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. A ***K-ary tree*** has approximately 1/K1/K of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal nodes, in a B+ tree of order 100 probably only about 1/751/75 of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

TIME ANALYSIS FOR INSERTING RECORD

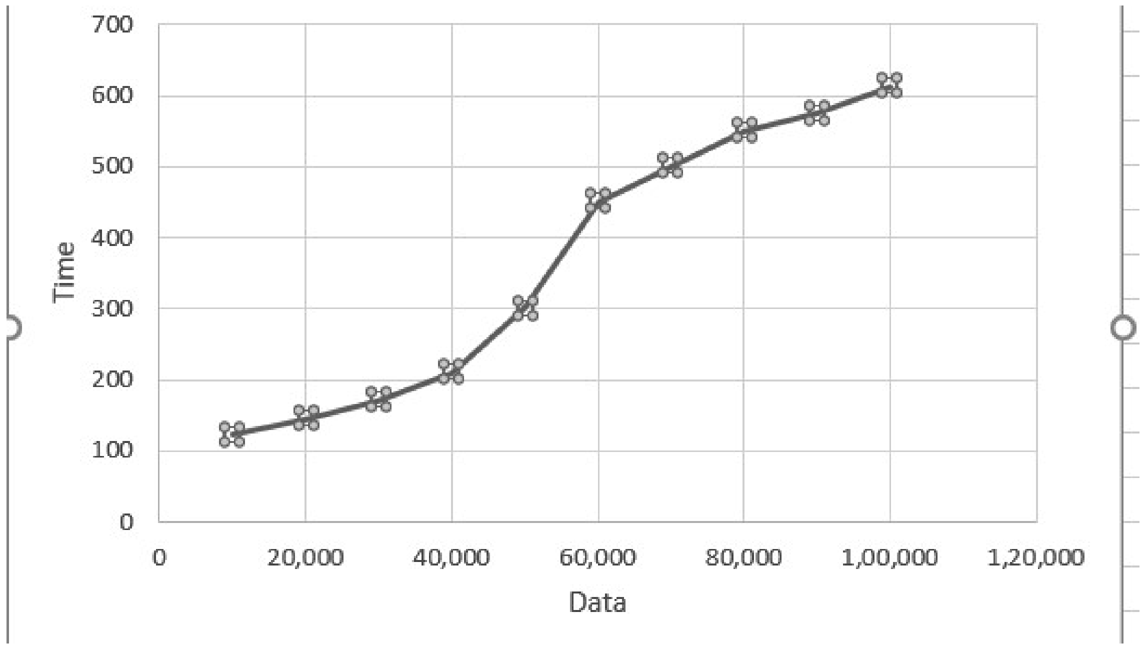


Fig 3.1: Time analysis for inserting the records:

The time taken to insert record into the datafile, the number of record Increases the time also increase into the datafile.

TIME ANALYSIS FOR SEARCHING RECORD

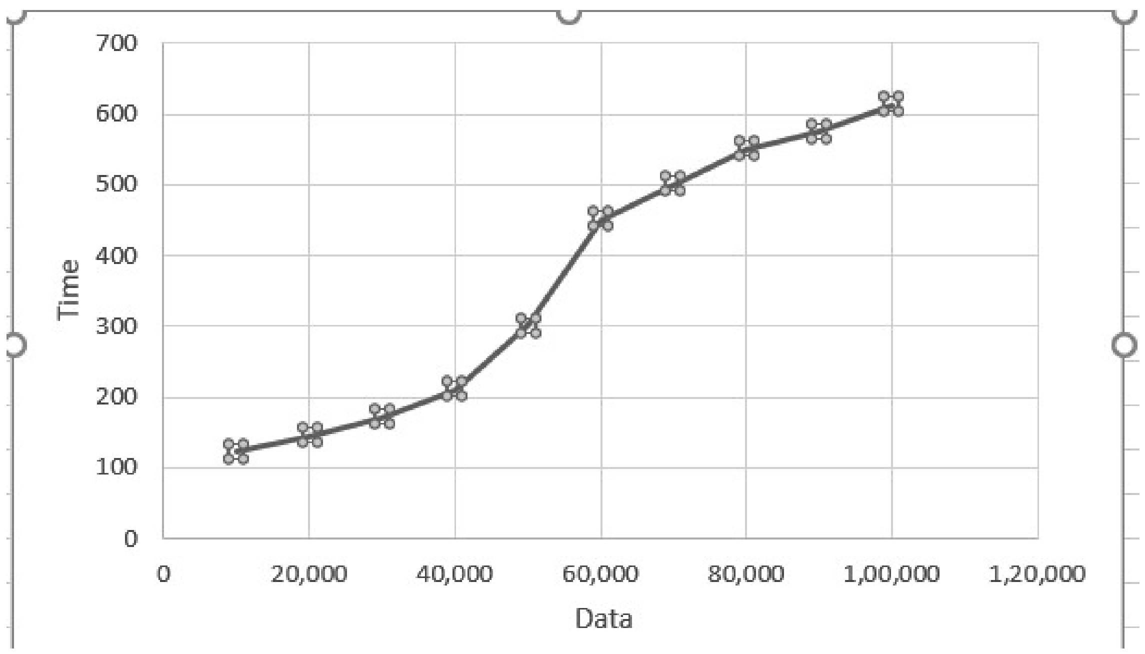


Fig3.2: Time analysis for the searching the record:

The time taken to search the record in the data file,as the number of records increases the time will be delayed to search the record

TIME ANALYSIS FOR B-TREE GENERATION

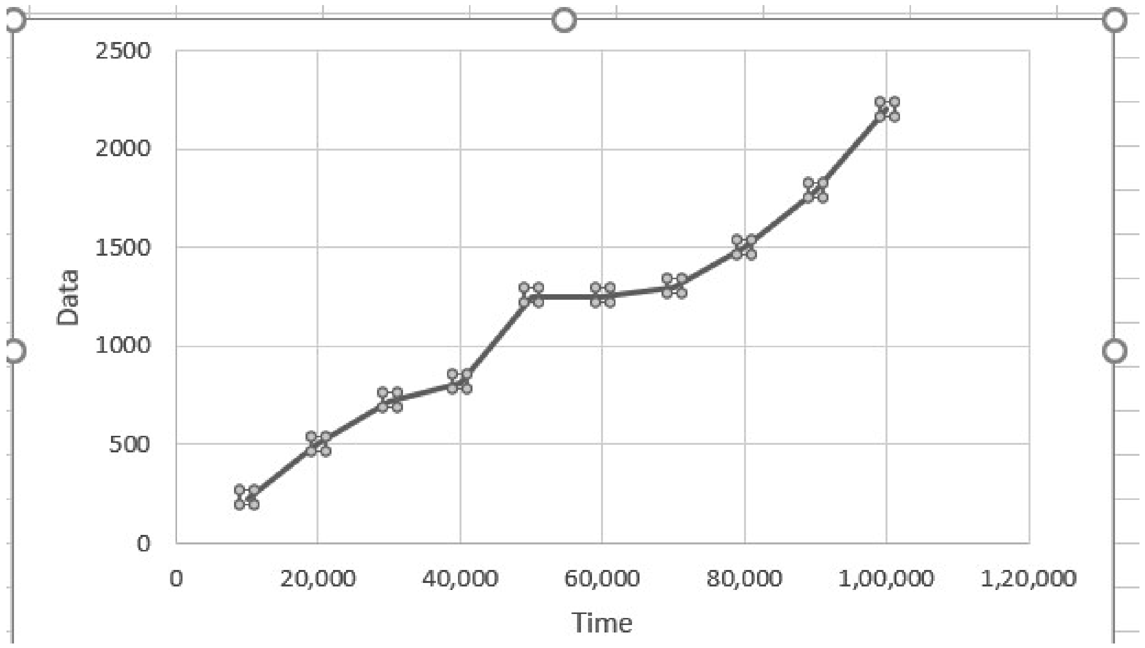
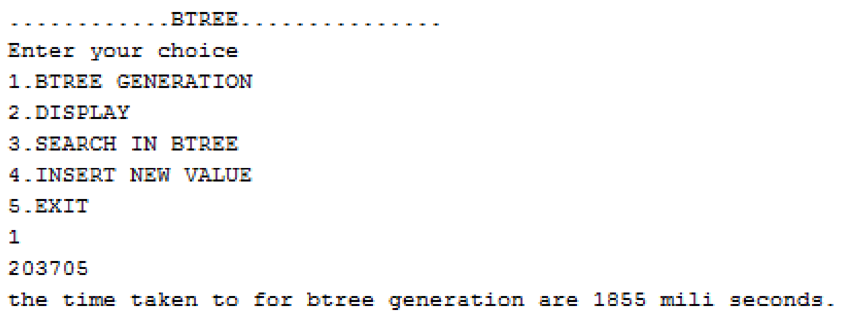


Fig3.3:Time analysis for the btree generation

Here we are obtaining graph based on the time taken for generation of B-tree for different set of records.

CHAPTER 4
RESULTS AND SCREENSHOTS

1. GENERATION OF B-TREE

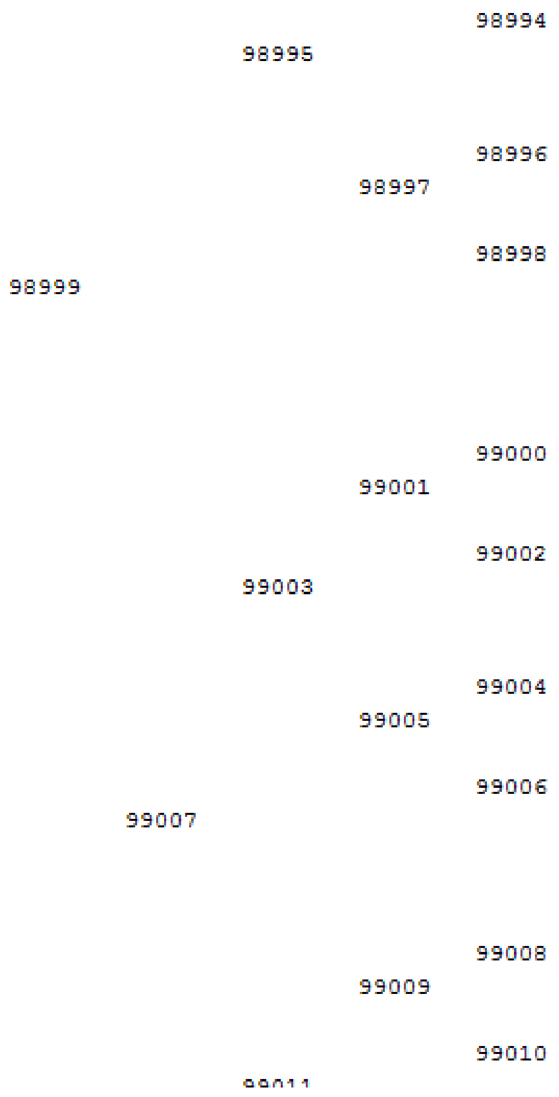


```
............BTREE...............
Enter your choice
1.BTREE GENERATION
2.DISPLAY
3.SEARCH IN BTREE
4.INSERT NEW VALUE
5.EXIT
1
203705
the time taken to for btree generation are 1855 mili seconds.
```

Fig 4.1 Generation of B-tree

The above shown snapshot describes about the time taken for generation of B-tree. The time taken for generation of B-tree for 1 lakh record is 1855ms**.**

2. DISPLAY OF B-TREE

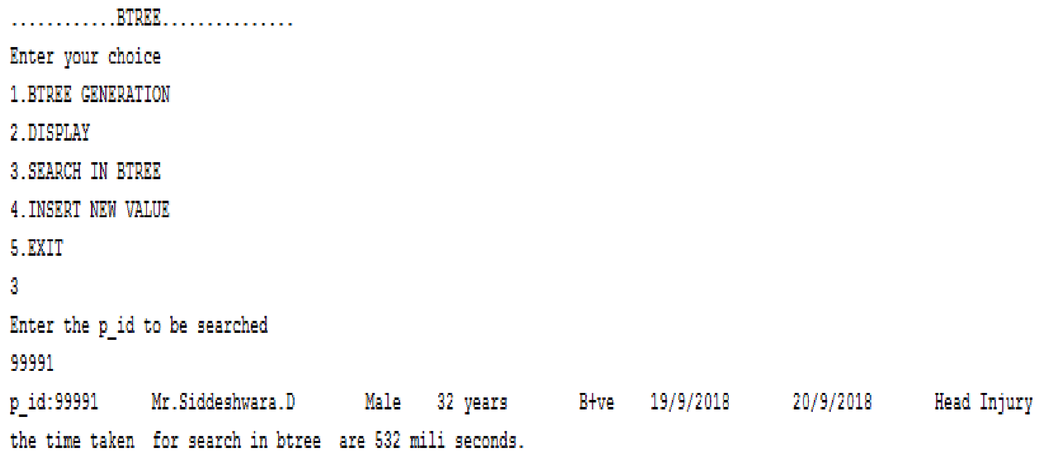


```
                                        98994
                    98995


                                        98996
                              98997

                                        98998
          98999




                                        99000
                              99001

                                        99002
                    99003


                                        99004
                              99005

                                        99006
               99007




                                        99008
                              99009

                                        99010
                    99011
```

Fig 4.2 Display of B-tree

The above shown snapshot is the B-tree stucture build by cosidering the primary key of 1 lakh records.Here in this case,p_id is considered as primary key for construction of B-tree.

3.SEARCH IN B-TREE

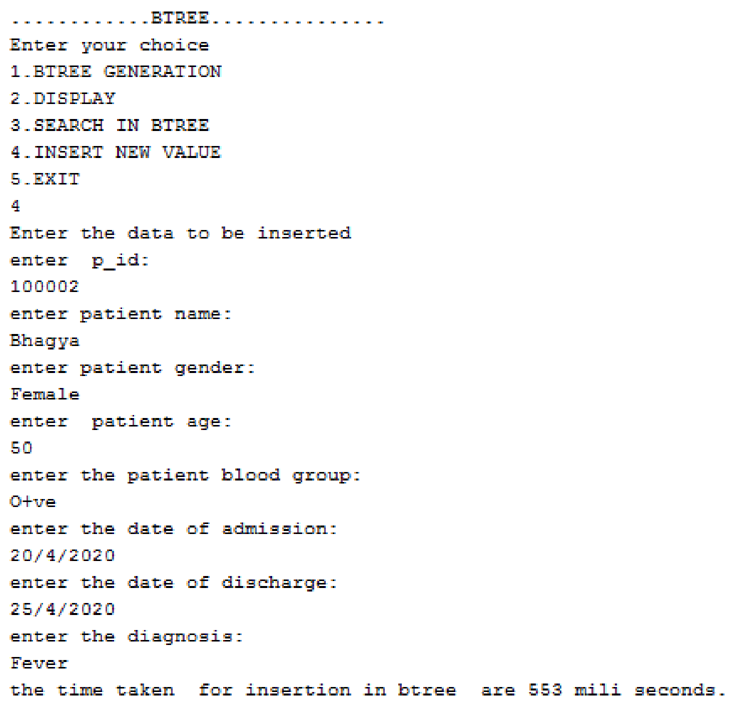


```
...........BTREE...............
Enter your choice
1.BTREE GENERATION
2.DISPLAY
3.SEARCH IN BTREE
4.INSERT NEW VALUE
5.EXIT
3
Enter the p_id to be searched
99991
p_id:99991     Mr.Siddeshwara.D       Male    32 years       B+ve    19/9/2018       20/9/2018       Head Injury
the time taken  for search in btree  are 532 mili seconds.
```

Fig 4.3 Search in B-tree

The above shown snapshot describes the working of a search function.Here in this case, we are fetching a particular hospital record by refering to the id which is the primary key for the hospital record. The time taken for searching a specified record is also been calculated.Ex:The time taken for searching a record with patient_id:99991 is 532ms

4.INSERTION OF NEW VALUES INTO B-TREE

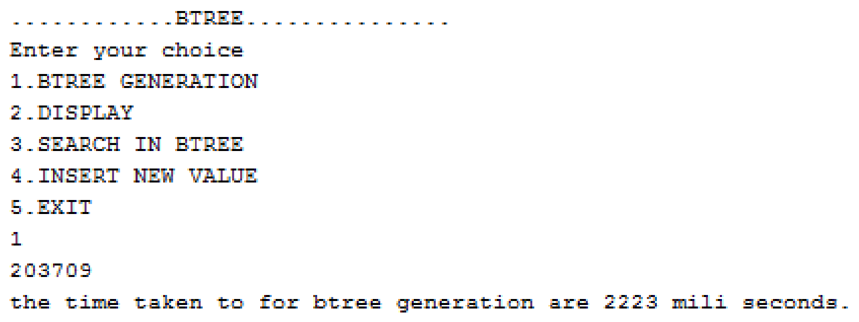


```
............BTREE...............
Enter your choice
1.BTREE GENERATION
2.DISPLAY
3.SEARCH IN BTREE
4.INSERT NEW VALUE
5.EXIT
4
Enter the data to be inserted
enter  p_id:
100002
enter patient name:
Bhagya
enter patient gender:
Female
enter  patient age:
50
enter the patient blood group:
O+ve
enter the date of admission:
20/4/2020
enter the date of discharge:
25/4/2020
enter the diagnosis:
Fever
the time taken  for insertion in btree  are 553 mili seconds.
```

Fig 4.4 Insertion in B-tree

The above shown snapshot describes about insertion of a record. The attributes for hospital record are patient id(p_id), patient name, patient gender, patient age ,patient blood group, date of admission, date of discharge and diagnosis . The time taken insertion of a record is also been calculated.

5. TIME TAKEN TO GENERATE B-TREE AFTER INSERTION OF NEW VALUE



```
............BTREE...............
Enter your choice
1.BTREE GENERATION
2.DISPLAY
3.SEARCH IN BTREE
4.INSERT NEW VALUE
5.EXIT
1
203709
the time taken to for btree generation are 2223 mili seconds.
```

Fig 4.5 B-tree generation

The above shown snapshot describes the time taken for generation of B-tree after insertion of a record .

CONCLUSION

I have successfully implemented B Tree which helps us in administrating the data used for managing the tasks performed.

We have successfully used various functionalities of JAVA and created the File structures.

Features:

1. Clean separation of various components to facilitate easy modification and revision.
2. All the data is maintained in a separate file to facilitate easy modification
3. All the data required for different operations is kept in a separate file.
4. Quick and easy saving and loading of database file.

REFERENCES

The information about B-tree was gathered by referring to the following sites**:**

- Github(github.com)
- Wikipedia(www.wikipedia.org)
- Stackoverflow(stackoverflow.com)
- We3schools(Wethreeschools.com)
- GeeksforGeeks(GeeksforGeeks.com)
- Java Debugger(javadebugger.com)